GeneSyst: a Tool to Reason about Behavioral Aspects of B Event Specifications. Application to Security Properties*

Didier Bert, Marie-Laure Potet, and Nicolas Stouls

Laboratoire Logiciels Systèmes Réseaux - LSR-IMAG - Grenoble, France
{Didier.Bert, Marie-Laure.Potet, Nicolas.Stouls}@imag.fr

Abstract. In this paper, we present a method and a tool to build symbolic labelled transition systems from B specifications. The tool, called GeneSyst, can take into account refinement levels and can visualize the decomposition of abstract states in concrete hierarchical states. The resulting symbolic transition system represents all the behaviors of the initial B event system. So, it can be used to reason about them. We illustrate the use of GeneSyst to check security properties on a model of electronic purse.

1 Introduction

Formal methods, such as the B method [1], ensure that the development of an application is reliable and that properties expressed in the model are satisfied by the final program. However, they do not guarantee that this program fulfills the informal requirements, nor the needs of the customer. So, it is useful to propose several views about the specifications, in order to be sure that the initial model is suitable for the customer and that the development can continue on this basis. One of these important insights is the representation of the behavior of programs by means of diagrams (statecharts). Moreover, some particular views, if they are themselves formal, can provide new means to prove properties that cannot easily be checked in the first model.

In this paper, we present a method and a tool to extract a labelled transition system from a model written in event-B. The transition system gives a graphical view and represents symbolically all the behaviors of the B model. The method is able to take into account refinement levels and to show the correspondence between abstract and concrete systems, by means of hierarchical states.

We present also an application of this tool, namely, the verification of security properties. The security properties assert the occurrence or the absence of some particular events in some situation. They are a case of *atomicity property* of transactions. This is illustrated by an example of specification of an electronic purse, called Demoney[16, 15], developed in the SecSafe project [19]. This

* This work was done in the GECCOO project of program "ACI : Sécurité Informatique" supported by the French Ministry of Research and New Technologies. It is also suported by CNRS and ST-Microelectronics by the way of a doctoral grant.

case study, written in Java Card [21], is an applet that has all the facilities required by a real electronic purse. Indeed, the purse can be debited from a terminal in a shop, credited by cash or from a bank account with a terminal in a bank or managed from special terminal in bank restricted area. Transactions are encrypted if needed and different levels of security are used depending on the actions. Demoney also supports to communicate with another applet on the card, for example, to manage award points on a loyalty plan. The specification of Demoney is public in version 0.8 [16], but the source code is copyrighted by Trusted Logic S.A.[1].

In Section 2, we recall the main features of event-B systems and refinements. We introduce a notion of behavioral semantics by the way of sequences of events. In Section 3, we define symbolic labelled transition systems (SLTS) and the links between SLTS and event-B systems are stated. In Section 4, we present the GeneSyst tool and an example of generation of SLTS dealing with the error cases in the Demoney case study. Section 5 presents security properties required in the application and shows how the GeneSyst diagrams can be used to check these properties. Then, we review related works, and we conclude the paper with some research perspectives in Section 6.

2 Event-B

2.1 General presentation

Event-B was introduced by J.-R. Abrial [2, 3]. It is a formal development method as well as a specification language. In event-B, components are composed of constant declarations (SETS, CONSTANTS, PROPERTIES), state specification (VARIABLES, INVARIANT), initialisation and set of *events*. The events are defined by $e \mathrel{\widehat{=}} eBody$ where e is the name of the event and $eBody$ is a *guarded* generalized substitution [1]. The events do not take parameters and do not return result values. They do not get preconditions and do terminate. Their effect is only to modify the internal state. If $\mathcal{S}$ is a component, then we denote by $Interface(\mathcal{S})$ the set of its events.

A well-typed and well-defined component is consistent if initialization $Init$ establishes the invariant of the component and if each event preserves the invariant. So, using the notation $[S]R$ as the weakest precondition of R for substitution S, the consistency of a component is expressed by the proof obligations: $[Init]I$ and $I \Rightarrow [eBody]I$ for each event.

In the paper, we use the notions of before-after predicate of substitution T for variables x ($\mathsf{prd}_x(T)$) and the feasability predicate of a substitution as defined in the B-Book: $\mathsf{fis}(T) \Leftrightarrow \neg[T]false$ [1]. Finally, the notation $\langle T\rangle R$ means $\neg[T]\neg R$, that is to say, there exists a computation of T which terminates in a state verifying R.

[1] http://www.trusted-logic.fr/

2.2 Events and traces

The events have the form "$e \;\widehat{=}\; G \Longrightarrow T$" where G is a predicate, T is a generalized substitution such that $I \wedge G \Rightarrow \mathsf{fis}(T)$. Predicate G is called the *guard* of e and T is its *action*. They are respectively denoted by $Guard(e)$ and $Action(e)$. If the syntactic definition of an event $e \;\widehat{=}\; S$ does not fulfill this form, it can be built by computing $e \;\widehat{=}\; \mathsf{fis}(S) \Longrightarrow S$. Following the so-called *event-based* approach [10], the semantics of event-B systems can be chosen to be the set of all the valid sequences of event executions.

Definition 1 (Traces of Event-B systems) *A finite sequence of event occurrences $e_0.e_1.e_2 \ldots e_n$ is a trace of system $\mathcal{S}$ if and only if e_0 is the initialisation of $\mathcal{S}$, $\{e_1, e_2, \ldots, e_n\} \subseteq Interface(\mathcal{S})$ and $\mathsf{fis}(e_0 \;;\; e_1 \;;\; e_2 \;;\; \ldots \;;\; e_n) \Leftrightarrow true$.*

The set of all the finite traces of a system $\mathcal{S}$ is called $Traces(\mathcal{S})$. For the initialisation, one can notice that $\mathsf{prd}_x(Init)$ does not depend on the initial values of the variables and that $Guard(Init) \Leftrightarrow true$. The following property characterizes traces by the existence of intermediary states x_i in which the guard of e_i holds and where the pair (x_i, x_{i+1}) is in the before-after predicate of event e_i:

Property 1 (Trace characterization) *Let x be the variable space of system $\mathcal{S}$, then:* $e_0.e_1.\ldots.e_n \in Traces(\mathcal{S}) \;\Leftrightarrow$
$\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n} ([x := x_i]\, Guard(e_i) \wedge [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(Action(e_i)))$

2.3 Event-B refinement

In the event-B method, a refinement is a component called REFINEMENT. The variables can be refined (i.e. made more concrete) and a *gluing invariant* describes the relationship between the variables of the refinement and those of the abstraction. The events of refinement $\mathcal{R}$ must at least contain those of the abstraction $\mathcal{S}$ (i.e. $Interface(\mathcal{S}) \subseteq Interface(\mathcal{R})$). The other events are called *new* events.

We recall here the proof obligations of system refinements. Let I be the invariant of the abstraction $\mathcal{S}$ and J be the invariant of refinement $\mathcal{R}$, then the gluing invariant is the conjunction $I \wedge J$. The refinement is performed elementwise, that is to say, the abstract initialisation is refined by the concrete initialisation and each abstract event is refined by its concrete counterpart. Proof obligations that establish the consistency of refinements are :

For initialisation *Init* :	$[Init^R]\langle Init^S\rangle J$
For events e of $Interface(\mathcal{S})$:	$I \wedge J \Rightarrow [e^R]\langle e^S\rangle J$
For the new events ne^R :	$I \wedge J \Rightarrow [ne^R]\langle \mathsf{skip}\rangle J$

New events cannot indefinitely take the control, i.e. the refined system cannot diverge more often that the abstract one. So, a *variant* V is declared in the refined system, as an expression on a well-founded set (usually the natural numbers), and the new events must satisfy (v is a fresh variable) :

V is a natural expression : $I \wedge J \Rightarrow V \in \mathbb{N}$

New events ne^R decrease the variant : $I \wedge J \Rightarrow [v := V][ne^R](V < v)$

Finally, a proof obligation of *liveness preservation* is usually required. If $\mathcal{S}$ contains m events and $\mathcal{R}$ contains p new events, then:

$$I \wedge J \Rightarrow (\bigvee_{i=1}^{m} Guard(e_i^S) \Rightarrow (\bigvee_{i=1}^{m} Guard(e_i^R) \ \vee \ \bigvee_{i=1}^{p} Guard(ne_i^R)))$$

Traces associated to refinements are defined as for the systems.

3 Symbolic labelled transition systems associated to B systems

3.1 Symbolic transition systems

We define *symbolic* labelled transition systems:

Definition 2 (Symbolic labelled transition system) *A symbolic labelled transition system (SLTS) is a 4-uple $(N, Init, U, W)$ where*
- *N is a set of states, and $Init$ is the initial state ($Init \in N$)*
- *U is a set of labels of the form (D, A, e), where D and A are predicates and e is an event name*
- *W is a transition relation $W \subseteq \mathbb{P}(N \times U \times N)$.*

A transition $(E, (D, A, e), F)$ means that, in state E, the event e is enabled if D holds and, starting from state E, if event e is enabled, then it reaches state F if A holds. Predicate D is called the *enabledness* predicate and A is called the *reachability* predicate.

States N are interpreted as subsets of variable spaces on variables x. So, the interpretation of N is given by a function $\mathcal{I}$ such that $\mathcal{I}(E)$ is a predicate on free variables x which characterizes the subset represented by E. In the next definition, we determine the actual conditions to cross a transition from a particular state value x_1 of E_1 to x_2 of E_2 by an event e which is defined in an event-B system $\mathcal{S}$. For that, e must be enabled in x_1, x_2 must be reachable from x_1 by e, and (x_1, x_2) must belong to the before-after predicate of e:

Definition 3 (Transition crossing) *Let $(E_1, (D, A, e), E_2)$ be a transition of a SLTS $\mathcal{T}$ on a system $\mathcal{S}$, and given x_1 and x_2 some values of the state variables x which satisfy the invariant of $\mathcal{S}$, then a crossing from x_1 to x_2 by this transition is legal if and only if :*
1. $[x := x_1](\mathcal{I}(E_1) \wedge D \wedge A)$
2. $[x, x' := x_1, x_2]\, \mathsf{prd}_x(Action(e))$
3. $[x := x_2]\mathcal{I}(E_2)$

Such a legal transition crossing is denoted by :

$$(E_1, x_1) \rightsquigarrow^{(D,A,e)} \rightsquigarrow \ (E_2, x_2)$$

Now, we introduce the notion of *path* in a symbolic labelled transition system. A path is a sequence of event occurrences, starting from the initial state, which goes over a transition system through legal transition crossings.

Definition 4 (Paths) *Given a symbolic labelled transition system $\mathcal{T}$ on a system $\mathcal{S}$, a sequence of event occurrences $e_0.\ldots.e_{n+1}$ is a path in $\mathcal{T}$ if there exists a list of states $E_0, \ldots, E_{n+1}$ of N, with $E_0 = Init_{\mathcal{T}}$, and a list of transitions $(D_i, A_i, e_i), i \in 0..n$, such that :*

$$\exists x_0, \ldots, x_{n+1} \cdot (\bigwedge_{i=0}^{n}((E_i, x_i) \rightsquigarrow^{(D_i, A_i, e_i)} (E_{i+1}, x_{i+1})))$$

The set of all the finite paths of $\mathcal{T}$ is called $Paths(\mathcal{T})$.

3.2 Construction of states and transitions

The aim of this section is to show how to compute a SLTS, from an event-B system $\mathcal{S}$ and given a set of states N. First, to build the states N, consider a list of predicates $\{P_1, \ldots, P_n\}$ on the variable space. We require that this set is *complete* with respect to the invariant, i.e. all the states specified by the invariant are included in the states determined by the P_i predicates, i.e.

$$I \;\Rightarrow\; \bigvee_{i=1}^{n} P_i$$

Then, the states of the SLTS are $N = \{Init_{\mathcal{S}}, E_1, \ldots, E_n\}$ with the interpretation defined by:

$$\mathcal{I}(Init_{\mathcal{S}}) = true \qquad\qquad \mathcal{I}(E_i) = P_i \;\wedge\; I, \quad i \in 1..n$$

We denote by $N1$ the set $N - \{Init_{\mathcal{S}}\}$. From the completeness property above and the definition of N, we get: $I \;\Leftrightarrow\; \bigvee_{i=1}^{n} \mathcal{I}(E_i)$.

Now, we express the conditions to ensure that a symbolic labelled transition system $\mathcal{T}$ represents the same set of behaviors as the associated system $\mathcal{S}$. For that, in a starting state E, the enabledness condition must be equivalent to the guard of the event e, and if the target state is F, the reachability condition must be equivalent to the possibility to reach F through e, when the enabledness predicate holds, so the condition:

Condition 1 (Valid transitions) *Let $\mathcal{S}$ be a system, E and F two states in N as defined above, and e an event, then the transition (E,(D,A,e),F) is valid if and only if predicates D and A satisfy :*

a) $\mathcal{I}(E) \;\Rightarrow\; (D \Leftrightarrow Guard(e))$

b) $\mathcal{I}(E) \wedge Guard(e) \;\Rightarrow\; (A \Leftrightarrow \langle Action(e)\rangle \mathcal{I}(F))$

Notice that, by applying the definition of the conjugate weakest precondition, condition *b)* is equivalent to :

$$\mathcal{I}(E) \wedge Guard(e) \;\Rightarrow\; (A \Leftrightarrow \exists x' \cdot (\mathsf{prd}_x(Action(e)) \wedge [x := x']\mathcal{I}(F)))$$

A SLTS with all the transitions valid with respect to a system $\mathcal{S}$ is called a valid symbolic labelled transition system.

Theorem 1 (Traces and paths equality) *Let $\mathcal{S}$ be an event-*B *system with invariant I and events Ev and let $\mathcal{T}$ be a valid symbolic labelled transition system built from $\mathcal{S}$, then:*

$$Traces(\mathcal{S}) = Paths(\mathcal{T})$$

Proof: We prove that, for all t, $t \in Paths(\mathcal{T}) \Leftrightarrow t \in Traces(\mathcal{S})$.
The path $t \ \widehat{=}\ e_0.e_1.\ldots.e_n$ is a path for the state sequence $E_0, E_1, \ldots, E_{n+1}$ iff (Definition 4): $\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n}((E_i, x_i) \rightsquigarrow^{(D_i, A_i, e_i)} (E_{i+1}, x_{i+1}))$
By using Definition 3, we get:

$$\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n}([x := x_i](\mathcal{I}(E_i) \wedge D_i \wedge A_i)$$
$$\wedge\ [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(Action(e_i))\ \wedge\ [x := x_{i+1}]\mathcal{I}(E_{i+1}))$$

By Condition 1, one can replace D_i by $Guard(e_i)$ and A_i by $\exists x' \cdot (\mathsf{prd}_x(Action(e_i)) \wedge [x := x']\mathcal{I}(E_{i+1}))$. The formula above is simplified and becomes:

$$\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n}([x := x_i](\mathcal{I}(E_i) \wedge Guard(e_i)) \quad (1)$$
$$\wedge\ [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(Action(e_i))\ \wedge\ [x := x_{i+1}]\mathcal{I}(E_{i+1}))$$

We must prove that this formula is equivalent to the characterization of the traces (Property 1):

$$\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n}([x := x_i]Guard(e_i) \quad (2)$$
$$\wedge\ [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(Action(e_i))\ \wedge\ [x := x_{i+1}]I)$$

Implication (1) $\Rightarrow$ (2) is verified because states E_i are such that $\mathcal{I}(E_i) \Rightarrow I$ (Section 3.2). To prove (2) $\Rightarrow$ (1), we must exhibit a list of states $E_0, E_1, \ldots, E_{n+1}$ such that these states satisfy (1). This follows from the fact that $\mathcal{I}(E_0) = true$ and from $I \Rightarrow \bigvee_{i=1}^{n} \mathcal{I}(E_i)$, which ensures that one of the states $\mathcal{I}(E_i)$ necessarily holds when I hold. □

3.3 Labelled transition systems for the refinements

We propose now the construction of a symbolic labelled transition system for the refinements. Our aim is to highlight the links between abstract and concrete transition systems, while preserving the overall structure of the abstract system. One aspect of the refinement is the change of the variable representation and redefinition of the events of the abstraction, according to the new representation. The point is taken into account by the notion of *state projection*.

In the following, $\mathcal{S}$ is a specification, $\mathcal{R}$ is its refinement with gluing invariant L, and $\mathcal{T}^S$ is a symbolic labelled transition system for $\mathcal{S}$. States E^S and F^S are states in $\mathcal{T}^S$. We assume that the variable set x^S of $\mathcal{S}$ is disjoint to the variable set x^R of the refinement. If some variables of the specification are kept in the refinement, they can be renamed and an equality between both variables is added to the invariant.

Definition 5 (State projection) *Let $\mathcal{S}$ be a system with variables x^S and $\mathcal{R}$ be the refinement of $\mathcal{S}$ according to L. A state E^R of $\mathcal{T}^{\mathcal{R}}$, $E^R \neq Init_{\mathcal{R}}$ is the projection of E^S of $\mathcal{T}^{\mathcal{S}}$, denoted by $E^R = Proj_L(E^S)$, iff:*

$$\mathcal{I}(E^R) \Leftrightarrow \exists x^S \cdot (L\ \wedge\ \mathcal{I}(E^S))$$

We propose to build a SLTS, called $Proj_L(\mathcal{T}^S)$, in which states are automatically deduced from abstract states and gluing invariant. The SLTS projection $Proj_L(\mathcal{T}^S)$ of the refinement $\mathcal{R}$ of system $\mathcal{S}$ with gluing invariant L is such that: the initial state is any q_0 with $\mathcal{I}(q_0) = true$; the other states of the projection are the projections of abstract states, i.e. $N1^R = \{Proj_L(q) \mid q \in N1^S\}$. The transitions are $(E^R, (D', A', e^R), F^R)$ where $e^R \in \mathcal{R}$ and D', A' are such that Condition 1 is satisfied. A transition $(E^R, (D', A', e^R), F^R)$ is said a *projection of transition* $(E^S, (D, A, e^S), F^S)$ iff $E^R = Proj_L(E^S)$, $F^R = Proj_L(F^S)$ and event e^R is the refinement of e^S. By construction, $Paths(Proj_L(\mathcal{T}^S)) = Traces(\mathcal{R})$. This equality can be proved in the same way as in Theorem 1.

Property 2 (Transition projection) *With the definitions above, let* $(E^R, (D', A', e^R), F^R)$ *be the projection of transition* $(E^S, (D, A, e^S), F^S)$*, then we have:*

$$\mathcal{I}(E^S) \ \wedge \ L \ \wedge \ D' \Rightarrow D$$

This property says that any transition enabled from a state $Proj_L(E^S)$ in a refinement $\mathcal{R}$ actually must be enabled in specification $\mathcal{S}$ (if the refinement is proved correct). Property 2 can make the computation of the transitions simpler. Indeed, if $e \in Interface(\mathcal{S})$, then, for all the transitions (E^S, e, F^S) of the abstraction, it is only necessary to examine the transitions $(Proj_L(E^S), e, E')$ with $E' \in N1^R$. No other transition can be labelled by e from this state.

Another key aspect of refinement is the refinement of behaviors. New events may be introduced that make the actions more detailed. These new events are not observable at the abstract level, as the stuttering in TLA [11]. Very often, new variables are introduced. Thus, it is useful to visualize the states referring to these variable changes. In order to preserve the structure of the abstract system, we choose to refine each abstract state in an independent way. So, the transitions, relative to events which belong to $Interface(\mathcal{S})$, are preserved by the introduction of hierarchical states.

Definition 6 (Hierarchical states) *A set of sub-states* $\{E_1^R, \ldots, E_m^R\}$ *can be associated to a super-state* $Proj_L(E^S)$ *of* $\mathcal{R}$ *if and only if*

$$\bigvee_{i=1}^{m} \mathcal{I}(E_i^R) \Leftrightarrow \mathcal{I}(Proj_L(E^S))$$

In a refined system, the user must decide what projections of abstract states are decomposed and s/he must provide the predicates of the decomposition. If the abstract states are disjoint, then the transitions associated to the new events appear only between the sub-states of a hierarchical state. An example of refinement with decomposition of states is given in Section 4.4.

4 The GeneSyst tool

4.1 Presentation

The GeneSyst tool is intended to generate a symbolic labelled transition system $\mathcal{T}$ from an event-B system $\mathcal{S}$ and a set of states N. Such a generated SLTS

will be denoted by $\mathcal{T}(\mathcal{S}, N)$. The input of the tool is a B component, where the ASSERTIONS clause contains the formula $P_1 \vee \ldots \vee P_n$, which characterizes the list of predicates $\{P_1, \ldots, P_n\}$. By this way, the condition of completeness (section 3.2) is generated as proof obligation.

We give a sketch of the algorithm which computes the transitions of $\mathcal{T}(\mathcal{S}, N)$: it uses three main variables: the set of visited states, *visited*, the set of processed states, *processed*, and the set of computed transitions tr. First, the initial state is put in the *visited* set. Then each state E in the *visited* set is processed: this consists in computing the transitions $(E, (D, A, e), F)$ with all events e to all non-initial states F of the system. Predicates D and A are determined following the algorithm defined in the following section. If D or A are not *false* then the transition $(E, (D, A, e), F)$ is added to tr, and if F has not been processed, it is put in the *visited* set. After the processing of state E, E is removed from *visited* and put in set *processed*. When *visited* is empty, then tr contains all the computed transitions of $\mathcal{T}(\mathcal{S}, N)$ and *processed* contains the set of reachable states. The algorithm terminates, because the set of states to be visited is finite (bounded by the cardinal of N). This algorithm guarantees that the resulting SLTS is a valid transition system for $\mathcal{S}$, with given states N.

4.2 Proof obligations

A subprocedure of the algorithm is to determine effectively the enabledness predicate and the reachability predicate, given a triple (E, e, F). For sake of usability of the resulting transition system, it is interesting to examine three cases: predicates are *true*, *false* or other. This information can be obtained by proof obligations. In Fig. 1, we give the conditions for the calculus of these predicates. Obviously, if D and/or A is *false*, then the transition is not possible.

	Proof obligations	D for (E, e, F)
(1)	$\forall x \cdot (\mathcal{I}(E) \Rightarrow \mathit{Guard}(e))$	true
(2)	$\forall x \cdot (\mathcal{I}(E) \Rightarrow \neg \mathit{Guard}(e))$	false
(3)	$\exists x \cdot (\mathcal{I}(E) \wedge \mathit{Guard}(e))$	$\mathit{Guard}(e)$
	Proof obligations	A for (E, e, F)
(4)	$\forall x \cdot (\mathcal{I}(E) \wedge \mathit{Guard}(e) \Rightarrow \langle \mathit{Action}(e) \rangle \mathcal{I}(F))$	true
(5)	$\forall x \cdot (\mathcal{I}(E) \wedge \mathit{Guard}(e) \Rightarrow [\mathit{Action}(e)] \neg \mathcal{I}(F))$	false
(6)	$\exists x \cdot (\mathcal{I}(E) \wedge \mathit{Guard}(e) \wedge \langle \mathit{Action}(e) \rangle \mathcal{I}(F))$	$\langle \mathit{Action}(e) \rangle \mathcal{I}(F)$

Fig. 1. Proof obligations for enabledness and reachability

In practice, the GeneSyst tool computes the proof obligations (POs) above and interacts with AtelierB to discharge the POs. For each triple (E, e, F):

1. if proof obligation (1) is automatically discharged then D is *true*.
2. if proof obligation (2) is automatically discharged then D is *false* and transition (E, e, F) does not occur in the resulting $\mathcal{T}(\mathcal{S}, N)$.
3. otherwise, D is $\mathit{Guard}(e)$ by default.

Then, after cases 1. and 3., GeneSyst computes the proof obligations for determining the reachability predicate A.

4. if proof obligation (4) is automatically discharged then A is *true*.
5. if proof obligation (5) is automatically discharged then A is *false* and transition (E, e, F) does not occur in the resulting $\mathcal{T}(\mathcal{S}, N)$.
6. otherwise, the transition is kept with $\langle Action(e) \rangle \mathcal{I}(F)$ as A, by default.

We can notice that Condition 1 about the validity of the transitions is well satisfied by construction. The *by default cases* in 3. and 6. correspond to several possibilities. Either there exist values in state E for which the transition is crossable (guard of e is true and state F is reachable), or there are not (the guard is false or state F is not reachable). However, in both possibilities, these transitions are included in the resulting transition system. To manage this feature, we define the notion of *minimal* symbolic labelled transition system.

Definition 7 (Minimal SLTS) *A minimal SLTS is a SLTS where all the transitions are valid, i.e. satisfy a) and b) of Condition 1, and also satisfy:*

c) $D \not\Leftrightarrow false$ *and* $A \not\Leftrightarrow false$

A SLTS built by GeneSyst is minimal if all the proof obligations of D and A have been effectively discharged in step 1. or 2. and step 4. or 5. in the algorithm above. To minimize the number of by-default transitionss, we have designed two variants of the algorithm. The first optional alternative of the algorithm is to change cases 3. and 6. into:

3'. if proof obligation (3) is automatically discharged, then D is $Guard(e)$ by proof, otherwise, D is $Guard(e)$ by default.
6'. if proof obligation (6) is automatically discharged, then A is $\langle Action(e) \rangle \mathcal{I}(F)$ by proof, otherwise, the transition is kept with $\langle Action(e) \rangle \mathcal{I}(F)$ as A by default.

Another option of the tool allows the user to get the POs which have not been automatically discharged. Then, s/he can do an interactive proof to complete the work and return the information that the PO is discharged or not. However, the interactive mode is not very practicable when there are a great number of proof obligations that are not automatically discharged. It becomes useful to check actually the absence of some critical transitions (cases 2. and 4.).

4.3 Transition systems associated to the Demoney case study

In Fig. 2, we give an example of transition system generated from a subset of the abstract specification of the Demoney case study. The B machine is provided in appendix. We just have represented four methods imposed by the Demoney specification [16]: *InitializeTransaction*, *CompleteTransaction*, *Reset* and *GetData*. The two methods *InitializeTransaction* and *CompleteTransaction* have to be executed in sequence. If they are called in the wrong order then an error must

be returned. Moreover, any other methods cannot be invoked between them, except the method *Reset* which models the extraction of the card from the terminal. If it is called during a transaction, all the internal variables must be restored at their initial values. Finally, method *GetData* has been defined to represent any other method which plays a neutral rôle with respect to transactions.

Let us notice that our model has been expressed with events. In the applet Demoney, methods have neither parameters nor result, because they communicate through a global variable, named $APDU$, which allows the information transfert between the card and the terminal. An error can be returned by means of the same variable. Finally, methods have no precondition, because they are callable at any time. So the transformation of methods in events is straightforward.

In the diagrams generated by GeneSyst, transitions are prefixed by the information about predicates D and A. A predicate denoted by "[ ]" means *true*, while "[G]" means that the transition is computed by cases 3. or 6. (see section 4.2).

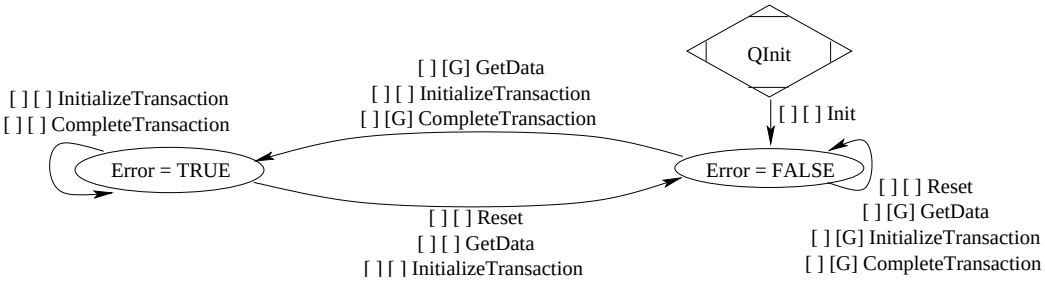

Fig. 2. Transition system associated to the error detection in the Demoney specification

Fig. 2 points out cases in which errors can occur. Transitions have no enabledness condition, because all the guards are *true* in the model. Some reachability conditions do not reduce to *true*, as for the event *GetData*, which is defined by:

$$\begin{aligned} GetData = \ & \text{IF } EngagedTrans = \mathsf{TRUE} \text{ THEN} \\ & \quad Error := \mathsf{TRUE} \parallel EngagedTrans := \mathsf{FALSE} \\ & \text{ELSE } Error := \mathsf{FALSE} \text{ END;} \end{aligned}$$

From state $Error = \mathsf{FALSE}$, event *GetData* can reach the state $Error = \mathsf{TRUE}$ with the condition $EngagedTrans = \mathsf{TRUE}$ and stays in $Error = \mathsf{FALSE}$ otherwise. Let us remark also that *GetData* is enabled in state $Error = \mathsf{TRUE}$ and always reaches state $Error = \mathsf{FALSE}$ because of the invariant $Error = \mathsf{TRUE} \Rightarrow EngagedTrans = \mathsf{FALSE}$.

4.4 Transition system associated to a refinement of Demoney

In our refinement of Demoney, the boolean variable *Error* is changed into a value of a given set *StatusType*, which intends to describe error codes, as imposed by the specification [16]. In the same way, the boolean variable *EngagedTrans* is refined into a value of a given set *TransactionType*, which indicates the exact

type of the current transaction. Finally, we have introduced the channel with two levels of security (FALSE and TRUE). All this information is declared in the invariant below (see also the refinement in appendix):

INVARIANT

$$
\begin{array}{l}
StatusWord \in StatusType \ \wedge \ CurTransaction \in TransactionType \ \wedge \\
ChannelIsSecured \in \mathsf{BOOL} \ \wedge \\
((Error = \mathsf{FALSE}) \Leftrightarrow (StatusWord = ISO\_Ok)) \ \wedge \\
((EngagedTrans = \mathsf{FALSE}) \Leftrightarrow (CurTransaction = None)) \ \wedge \\
((CurTransaction \neq None) \Rightarrow (ChannelIsSecured = \mathsf{TRUE})) \ \wedge \\
((StatusWord \neq ISO\_Ok) \Rightarrow (CurTransaction = None))
\end{array}
$$

Fig. 3 is built from this refinement. State $Error = \mathsf{FALSE}$, which corresponds to $StatusWord = ISO\_Ok$, is split into two states according to that a transaction is engaged or not.

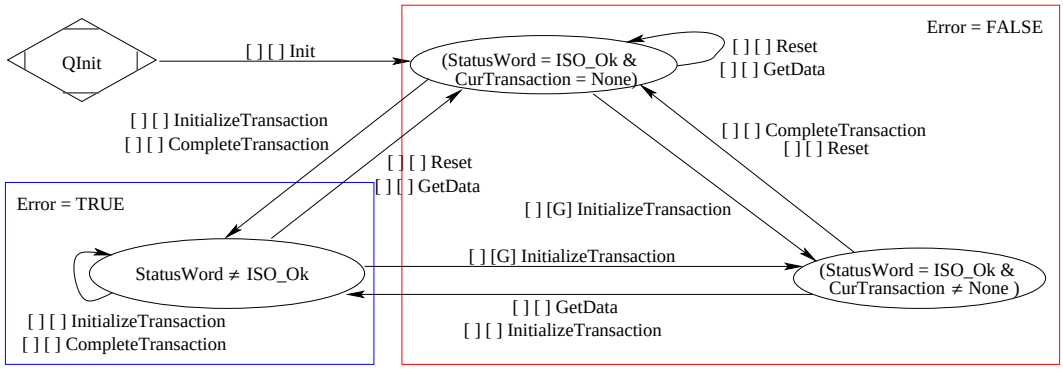

Fig. 3. Transition system associated to the refinement of the error detection

As expressed in Definition 6, the predicate given to GeneSyst to describe the states has to be a conjonction of equivalences between an abstract state and a disjonction of refined states. This predicate is written in the assertion clause. For example, the assertion below has been used to generate Fig. 3.

$$
\begin{array}{l}
((Error = \mathsf{TRUE}) \Leftrightarrow ((StatusWord \neq ISO\_Ok \ \wedge \ CurTransaction = None) \\
\qquad\qquad\qquad\qquad \vee \ (StatusWord \neq ISO\_Ok \ \wedge \ CurTransaction \neq None))) \\
\wedge \\
((Error = \mathsf{FALSE}) \Leftrightarrow ((StatusWord = ISO\_Ok \ \wedge \ CurTransaction = None) \\
\qquad\qquad\qquad\qquad \vee \ (StatusWord = ISO\_Ok \ \wedge \ CurTransaction \neq None)))
\end{array}
$$

With the splitting of the state $Error = \mathsf{FALSE}$, transition conditions are simplified in *true* or *false* or, in the worst case, are unchanged. For example, in Fig. 2, the transition labelled by $[][G]CompleteTransaction$ and going from $Error = \mathsf{FALSE}$ to $Error = \mathsf{TRUE}$ is, in Fig. 3, going from $StatusWord = ISO\_Ok \wedge CurTransaction = None$ to $StatusWord \neq ISO\_Ok \wedge CurTransaction = None$ with

the label [][]$CompleteTransaction$. So, its reachability has been made more precise. The same effect occurs on transition [][G]$CompleteTransaction$ going from $Error = \mathsf{FALSE}$ to $Error = \mathsf{FALSE}$, which is refined by [][]$CompleteTransaction$ going from $CurTransaction \neq None$ to $CurTransaction = None$ in the super-state $Error = \mathsf{FALSE}$. These two specializations are directly due to the introduction of the $CurTransaction$ variable.

5 Verification of Security Properties on Demoney

In this section we propose a formalism to express properties relative to security aspects and we show how GeneSyst can be used to verify these properties. We will next give a concrete example relative to the Demoney case study.

5.1 Properties

Generally, security is designed and implemented through different levels of abstraction. Security policies are defined by a set of rules according to which the system can be regulated, in order to guarantee expected properties, as confidentiality or integrity. Security policies are then implemented through software and hardware functions, called security mechanisms. Such an approach has been adopted by the **C**ommon **C**riteria norm [8] which proposes, through the notion of assurance requirements, a catalogue of security policies and a hierarchy of mechanisms.

In this paper we focus on security properties relative to constraints on the global behavior of the system, as authentication procedures or access control. In this case, security requirements can be seen as constraints on the execution order of atomic actions, as operation calls. F. Schneider claims in [18] that automata are a well-adapted formalism which can, both, be used to specify some forms of security policies and to control implementations during their execution. On the other hand, K. Trentelman and M. Huisman [22] propose a logic that can be used also to express some forms of security properties, as temporal properties on **JML** specifications.

We adopt a formalism based on logic formulas, which allows us to point out expected behaviors either in specifying correct executions, or in specifying security violations. That offers a good flexibility and is suitable to describe as well open policies as closed policies, respectively relative to negative authorizations and positive authorizations [17].

5.2 Predicates of security properties

Security properties are often represented as a list of first order logic formulas that have to be verified. We want to define some predicates to make the expression of these formulas easier. Predicates that we introduce express the ability of an event to start from a state (*Enabled* and *AlwaysEnabled*) and the existence of a transition between two states (*Crossable* and *AlwaysCrossable*).

Definition 8 (*Enabled*, *AlwaysEnabled*, *Crossable* and *AlwaysCrossable*) *Given p_1 and p_2 two state predicates and an event ev from a system $\mathcal{S}$ with variables x, then:*

$$\begin{array}{lcl} \mathit{Enabled}(p_1, ev) & \hat{=} & \exists x \cdot (p_1 \ \wedge \ \mathit{Guard}(ev)) \\ \mathit{AlwaysEnabled}(p_1, ev) & \hat{=} & \forall x \cdot (p_1 \Rightarrow \mathit{Guard}(ev)) \\ \mathit{Crossable}(p_1, ev, p_2) & \hat{=} & \exists x \cdot (p_1 \ \wedge \ \langle ev \rangle p_2) \\ \mathit{AlwaysCrossable}(p_1, ev, p_2) & \hat{=} & \forall x \cdot (p_1 \Rightarrow [ev] p_2) \end{array}$$

Let us note that if $\mathit{Enabled}(p_1, ev) \Leftrightarrow \mathit{false}$, then, for each predicate p_2, $\mathit{AlwaysCrossable}(p_1, ev, p_2)$ will be *true* instead of *false*, which is the intuitive value expected. In the same way, if p_1 is equivalent to *false* then *AlwaysEnabled* and *AlwaysCrossable* are always *true*. Moreover, we can notice that:

$$\mathit{Crossable}(p_1, ev, p_2) \Rightarrow \mathit{Enabled}(p_1, ev)$$

From this definition we can deduce the properties below, relative to the implication:

Property 3 *Given p_1, p_2 and p_3 three predicates and an event ev then:*

- *if* $p_1 \Rightarrow p_3$ *and* $\mathit{Enabled}(p_1, ev)$ *then* $\mathit{Enabled}(p_3, ev)$
- *if* $p_3 \Rightarrow p_1$ *and* $\mathit{AlwaysEnabled}(p_1, ev)$ *then* $\mathit{AlwaysEnabled}(p_3, ev)$
- *if* $p_1 \Rightarrow p_3$ *and* $\mathit{Crossable}(p_1, ev, p_2)$ *then* $\mathit{Crossable}(p_3, ev, p_2)$
- *if* $p_2 \Rightarrow p_3$ *and* $\mathit{Crossable}(p_1, ev, p_2)$ *then* $\mathit{Crossable}(p_1, ev, p_3)$
- *if* $p_3 \Rightarrow p_1$ *and* $\mathit{AlwaysCrossable}(p_1, ev, p_2)$ *then* $\mathit{AlwaysCrossable}(p_3, ev, p_2)$
- *if* $p_2 \Rightarrow p_3$ *and* $\mathit{AlwaysCrossable}(p_1, ev, p_2)$ *then* $\mathit{AlwaysCrossable}(p_1, ev, p_3)$

Here are two examples:

Reactivity of a system. The **JavaCard** specification imposes that any APDU instruction is callable at any time. Given $\mathcal{S}$ a system and I its invariant, then this formula can be expressed as follows:

$$\forall ev \cdot (ev \in \mathit{Interface}(\mathcal{S}) \Rightarrow \mathit{AlwaysEnabled}(I, ev))$$

Unicity of the ways to reach a state. In some cases, like access control, we want to impose that the only way to reach a state P is to execute a particular event *Begin*. If I is the invariant of $\mathcal{S}$, then this property can be expressed as follows:

$$\forall ev \cdot (ev \in \mathit{Interface}(\mathcal{S}) \ \wedge \ ev \neq \mathit{Begin} \Rightarrow \mathit{AlwaysCrossable}(I, ev, \neg P))$$

5.3 Property checking using GeneSyst SLTS

Security properties could be verified on B specifications, using definition 8. Nevertheless, in some cases, the SLTS produced by GeneSyst can be directly exploited. Then, the verification consists in using syntactic information relative to enabledness and reachability of transitions. Properties 4–7 list the different cases where the predicates above can be directly established from a symbolic labelled transition system.

Properties 4–7 share the following hypothesis: *Given an event e and q_1, q_2 two states from a SLTS $\mathcal{T}$, such as $\mathcal{I}(q_1) \not\Rightarrow false$ and $(q_1, (D, A, e), q_2) \in W_{\mathcal{T}}$, then predicates Enabled, AlwaysEnabled, Crossable and AlwaysCrossable can be determined as follows:.*

Property 4 (Enabledness condition - general case)

1. $D \equiv true$	$\Rightarrow$	$Enabled(q_1, e)$
2. $D \equiv false$	$\Rightarrow$	$\neg Enabled(q_1, e)$
3. $D \equiv true$	$\Rightarrow$	$AlwaysEnabled(q_1, e)$
4. $D \equiv false$	$\Rightarrow$	$\neg AlwaysEnabled(q_1, e)$

If the SLTS used to verify the property is minimal, then Property 4 can be enlarged: the conditions are necessary (and sufficient) and conditions 1 and 4 are refined.

Property 5 (Enabledness for minimal SLTS)

1. $D \not\equiv false$	$\Leftrightarrow$	$Enabled(q_1, e)$
2. $D \equiv false$	$\Leftrightarrow$	$\neg Enabled(q_1, e)$
3. $D \equiv true$	$\Leftrightarrow$	$AlwaysEnabled(q_1, e)$
4. $D \not\equiv true$	$\Leftrightarrow$	$\neg AlwaysEnabled(q_1, e)$

In the same way, syntactic conditions to check *Crossable* and *AlwaysCrossable* predicates are:

Property 6 (Reachability condition - general case)

5. $A \equiv true$	$\Rightarrow$	$Crossable(q_1, e, q_2)$
6. $A \equiv false \ \vee \ D \equiv false$	$\Rightarrow$	$\neg Crossable(q_1, e, q_2)$
7. $A \equiv true \ \wedge$ $\forall q_i \cdot (q_2 \not\equiv q_i \Rightarrow (q_1, (D, A_2, e), q_i) \notin W_{\mathcal{T}})$	$\Rightarrow$	$AlwaysCrossable(q_1, e, q_2)$
8. $A \equiv false$	$\Rightarrow$	$\neg AlwaysCrossable(q_1, e, q_2)$

Cases 7 and 8 are not symetric, as it would be expected, because, syntacticaly, we can just compare names of states, not the intersection of their interpretation. Just as for enabledness, the conditions can be enlarged, when the SLTS is minimal, as follow:

Property 7 (Reachability for minimal SLTS)

5. $A \not\equiv false$	$\Leftrightarrow$	$Crossable(q_1, e, q_2)$
6. $A \equiv false \ \vee \ D \equiv false$	$\Leftrightarrow$	$\neg Crossable(q_1, e, q_2)$
8. $A \not\equiv true$	$\Rightarrow$	$\neg AlwaysCrossable(q_1, e, q_2)$

Cases 7 and 8 are just sufficient conditions because of the limitation of the syntactic verification. Case 7 is not present in Property 7 because it is the same as in Property 6 Finally, Property 3 allows the deduction of derived properties from the four properties above, by weakening or strenghtening the states.

5.4 Example of a property checking

In this section, we develop a real example of Demoney property and we do its verification by using the SLTS given in Figure 3. In the Demoney specification [16], the two APDU instructions *InitializeTransaction* and *CompleteTransaction* have to be executed in sequence, without any other instructions between them and without reaching any error state, to make a transaction. However, the card can be withdrawn at any time (modelled by the *Reset* event) without generating any error. Transaction atomicity property can be decomposed in five formulas given below, where I stands for the invariant of the Demoney specification. Moreover, SLTS of Figure 3 is minimal and events are always enabled from all state of the SLTS. Finally, note than the invariant I is equivalent to the union of all state predicates (Section 3.2).

Formula 1: There exists at least a value in I such that the event *InitializeTransaction* can reach $CurTransaction \neq None$:

$$Crossable(I, InitializeTransaction, CurTransaction \neq None)$$

Predicate $CurTransaction \neq None$ directly corresponds to a state predicate. Since there exists a transition from $CurTransaction = None \ \wedge \ StatusWord = ISO\_Ok$ to $CurTransaction \neq None$, labelled with $[\,][G]InitializeTransaction$, then we can use case 5 of Property 7 and conclude that the Formula 1 is *true*.

Formula 2: For all values, the event *InitializeTransaction* goes into the state $CurTransaction \neq None$ or into an error state:

$$AlwaysCrossable(I, InitializeTransaction, \\ CurTransaction \neq None \ \vee \ StatusWord \neq ISO\_Ok)$$

$CurTransaction \neq None$ and $StatusWord \neq ISO\_Ok$ are two state predicates, and all the transitions labelled with *InitializeTransaction* go only in one of these states. Then, due to case 7 of property 6, this formula is *true*.

Formula 3: From $CurTransaction \neq None$, all events, but $CompleteTransaction$ and *Reset*, go to an error state:

$$\forall e \cdot (e \in Interface(\mathcal{S}) \ \wedge \ e \neq CompleteTransaction \ \wedge \ e \neq Reset \Rightarrow \\ AlwaysCrossable(CurTransaction \neq None, e, StatusWord \neq ISO\_Ok)$$

The two predicates correspond to state predicates and the only events which go elsewhere than $StatusWord \neq ISO\_Ok$ from $CurTransaction \neq None$ are *CompleteTransaction* and *Reset*. Thus Formula 4 is *true* (case 7 of Property 6).

Formula 4: Except *InitializeTransaction*, no event can reach *CurTransaction* $\neq None$:

$$\forall e \cdot (e \in Interface(\mathcal{S}) \ \wedge \ e \neq InitializeTransaction \Rightarrow \\ AlwaysCrossable(I, e, CurTransaction = None))$$

Predicate $CurTransaction = None$ is the union of two existing state predicates. So, we have to check if there exists an event, different from *InitializeTransaction*, that can reach $CurTransaction \neq None$. Since it is not the case, this formula is *true* (case 7 of Property 6).

Formula 5: No transition labelled by *CompleteTransaction* or *Reset* is reflexive on state $CurTransaction \neq None$:

$\neg Crossable(CurTransaction \neq None, CompleteTransaction, CurTransaction \neq None)$

and $\neg Crossable(CurTransaction \neq None, Reset, CurTransaction \neq None)$

$CurTransaction \neq None$ corresponds to a state predicate and no *CompleteTransaction* or *Reset* reflexive transition occurs. Thus this formula is *true* (case 5 of Property 7).

The model of Demoney is thus correct relatively to the atomicity security property of transactions. However, during the realisation of this example, which is a simplified Demoney applet, we found three errors due to an erroneous simplification of our complete model of Demoney.

The originality of this approach is to have brought back, under some hypotheses, the verification of security properties to a syntactic checking. However, it is important to be careful about the real value of the crossing conditions generated by GeneSyst. Indeed, if some proof obligations are not (automatically) discharged, the transitions system will have by-default transitions. Then, to properly exploit the information, we have to be sure that the property to be verify can be checked on a non-minimal SLTS.

6 Related works and Conclusion

The work presented here is in line with the ideas presented in [5], itself inspired by [9]. In [5], the authors propose the construction of a labelled transition system which is a finite state abstraction of the behavior of an event-B system. The existence of transitions is determined by proof obligations, as here, but the resulting transition system does not contain any information about transition crossing. Moreover, the paper does not consider the refinement step in the diagram representation.

Other work is devoted to the translation of dynamic aspects described by statecharts in the B formalism (for instance [13, 20]). These approaches are inverse of ours, because they go from a diagrammatic representation to an encoding in a formal text. Their objective is to build a B model from UML descriptions. On our side, we suppose that the model has been stated and we are interested in representing the precise behavior of the system with respect to (a part of) variables, in order to check properties, or to validate the model against the requirements.

A similar approach has been envisaged for TLA [12] and extended in [6, 7] to take in account liveness properties and refinement. As in [5], the generated diagrams are abstractions of the system behavior.

Several tools are dedicated to the analysis of the behavior of B components by the way of the animation of machines [4] or by local exhaustive model checking [14]. Even if some of them allow the generation of symbolic traces, these tools can

be considered as "testing" tools. They provide particular execution sequences of the system, not a static representation of all the behaviors. In [23], the authors describe the generation of statecharts from event-B systems, but their approach suffers from several restrictions and their diagrams are not symbolic.

In this paper, we have presented the GeneSyst tool, its logical foundations and its application to the verification of security properties. In the first part, we introduced the definition of traces of event-B systems and refinements. We formalized the notion of symbolic labelled transition systems, with transitions decorated by enabledness and reachability predicates. This gives a complete and precise view of the behavior of the system, which can be exploited for various objectives.

We described the algorithm that is implemented to generate a SLTS from a B system and a set of states, characterized by predicates. The computation of effective transitions between states is performed by proving proof obligations. Due to the indecidability of the proof process, we have the choice between two kinds of (non exclusive) results: the generation is automatic, but we can get more transitions than in the real system, or the user completes interactively the non-conclusive proofs and then, the resulting automaton reflects exactly the behavior of the system.

The user can take profit of the freedom degree achieved by the choice of the states, to obtain the best analyses useful for him/her purpose. Non classical verification techniques can be designed and implemented at this stage, to assess or to validate the model, as it was shown in the last part of the paper. This opens a large field of research in the domains of security properties, confidentiality, access control, validation of models with respect to the requirements, automatic documentation of specifications, etc. Our present research work is to develop a set of techniques in the GECCOO[2] project to express and to check security properties, as it was sketched in the paper. We want to investigate the extraction of states from the specification of property automata, the use of refinement to split states and achieve a suitable level of decomposition in order to check a property. Another work is to deal with complex B models (several refinement chains together with composition clauses SEES, INCLUDES, etc.), either by composing partial labelled transition systems, or by flattening a structured model before computing the whole associated SLTS.

References

1. J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.
2. J.-R. Abrial. Extending B without Changing it (for Developing Distributed Systems). In H. Habrias, editor, *First B conference*, Putting into Practice Methods and Tools for Information System Design, IRIN, pages 169–191, 1996.

[2] "Génération de Code Certifié Orienté Objet". Project of Program "ACI Sécurité Informatique", 2003.

3. J.R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, LNCS 1393, pages 83–128. Springer-Verlag, 1998.
4. F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing tools: A tool-set for test generation from Z and B using constraint logic programming. In *Formal Approaches to Testing of Software (FATES'02)*, pages 105–120. INRIA, 2002.
5. D. Bert and F. Cave. Construction of Finite Labelled Transition Systems from B Abstract Systems. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods*, LNCS 1945, pages 235–254. Springer-Verlag, 2000.
6. D. Cansell, D. Méry, and S. Merz. Predicate Diagrams for the Verification of Reactive Systems. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods*, LNCS 1945, pages 380–397. Springer-Verlag, 2000.
7. D. Cansell, D. Méry, and S. Merz. Diagram Refinements for the Design of Reactive Systems. *Journal of Universal Computer Science*, 7(2), 2001.
8. Common Criteria. *Common Criteria for Information Technology Security Evaluation, Norme ISO 15408 - version 2.1*, Aout 1999.
9. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Computer-Aided Verification (CAV'97)*, LNCS 1254. Springer-Verlag, 1997.
10. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
11. L. Lamport. A Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, may 1994.
12. L. Lamport. TLA in Pictures. *Software Engineering*, 21(9):768–775, 1995.
13. H. Ledang and J. Souquières. Contributions for Modelling UML State-charts in B. In M. Butler, L. Petre, and K. Sere, editors, *IFM*, LNCS 2335, pages 109–127. Springer-Verlag, 2002.
14. M. Leuschel and M. Butler. ProB: A Model Checker for B. In K. Akari, S. Gnesi, and D Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 1997.
15. R. Marlet. DEMONEY: Java Card Implementation. Public technical report, SECSAFE project, 11 2002.
16. R. Marlet and C. Mesnil. DEMONEY : A demonstrative Electronic Purse - Card Specification -. Public technical report, SECSAFE project, 11 2002.
17. P. Samarati and S. De Capitani di Vimercati. Access Control: Policies, Models, and Mechanisms. In *Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design*, pages 137–196. Springer-Verlag, 2001.
18. F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.
19. SecSafe. SecSafe Porject Home Page. http://www.doc.ic.ac.uk/ siveroni/secsafe/.
20. E. Sekerinski and R. Zurob. Translating Statecharts to B. In M. Butler, L. Petre, and K. Sere, editors, *IFM*, LNCS 2335, pages 128–144. Springer-Verlag, 2002.
21. SUN. Java Card 2.1 Platform Specifications. http://java.sun.com/products/javacard/specs.html.
22. K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In *Algebraic Methodology And Software Technology (AMAST '02)*, LNCS 2422, pages 334–348. Springer-Verlag, 2002.
23. J.-C. Voisinet and B. Tatibouet. Generating Statecharts from B Specifications. In *16th Int Conf. on Software and System Engineering and their applications (ISCEA 2003)*, volume 1, 2003.

Appendix

Machine of the Demoney specification (diagram in Fig. 2, Section 4.3):

```
MACHINE Demoney
VARIABLES
    Error, EngagedTrans
INVARIANT
    Error ∈ BOOL ∧ EngagedTrans ∈ BOOL ∧
    (Error = TRUE ⇒ EngagedTrans = FALSE) ∧
    (EngagedTrans = TRUE ⇒ Error = FALSE)
ASSERTIONS
    /* The assertion provides the states for tool GeneSyst */
    /* Here, only two states are considered according to the Error values */
    Error = FALSE ∨ Error = TRUE
INITIALISATION
    Error := FALSE || EngagedTrans := FALSE
OPERATIONS
    Reset = BEGIN EngagedTrans := FALSE || Error := FALSE END;
    GetData =
        IF EngagedTrans = TRUE THEN
            Error := TRUE || EngagedTrans := FALSE
        ELSE Error := FALSE
        END;
    InitializeTransaction =
        IF EngagedTrans = TRUE THEN
            Error := TRUE || EngagedTrans := FALSE
        ELSE
            ANY SW WHERE SW ∈ BOOL THEN
                Error := SW || EngagedTrans := bool(SW = FALSE)
            END
        END;
    CompleteTransaction =
        IF EngagedTrans = FALSE THEN
            Error := TRUE
        ELSE Error := FALSE || EngagedTrans := FALSE
        END
END
```

Refinement of the Demoney specification (diagram in Fig. 3, Section 4.4):

```
REFINEMENT Demoney_R1
REFINES Demoney
SETS
    TransactionType = {Credit, Debit, None};
    StatusType = {ISO_Error, ISO_Ok}
VARIABLES
    StatusWord, CurTransaction, ChannelIsSecured
```

INVARIANT

$StatusWord \in StatusType \ \wedge \ CurTransaction \in TransactionType \ \wedge$
$ChannelIsSecured \in \mathsf{BOOL} \ \wedge$
$((StatusWord = ISO\_Ok) \Leftrightarrow (Error = \mathsf{FALSE})) \ \wedge$
$((EngagedTrans = \mathsf{TRUE}) \Leftrightarrow (CurTransaction \neq None)) \ \wedge$
$((CurTransaction \neq None) \Rightarrow ChannelIsSecured = \mathsf{TRUE}) \ \wedge$
$((StatusWord \neq ISO\_Ok) \Rightarrow (CurTransaction = None))$

ASSERTIONS

/* Each abstract state is decomposed in two concrete states */
/* One of these states is not reachable */
$((Error = \mathsf{TRUE}) \Leftrightarrow$
$((StatusWord \neq ISO\_Ok \ \wedge \ CurTransaction = None)$
$\vee \ (StatusWord \neq ISO\_Ok \ \wedge \ CurTransaction \neq None)))$
$\wedge$
$((Error = \mathsf{FALSE}) \Leftrightarrow$
$((StatusWord = ISO\_Ok \ \wedge \ CurTransaction = None)$
$\vee \ (StatusWord = ISO\_Ok \ \wedge \ CurTransaction \neq None)))$

INITIALISATION

$StatusWord := ISO\_Ok \ || \ ChannelIsSecured := \mathsf{FALSE} \ ||$
$CurTransaction := None$

OPERATIONS

$Reset =$ BEGIN
$StatusWord := ISO\_Ok \ || \ ChannelIsSecured := \mathsf{FALSE} \ ||$
$CurTransaction := None$
END;

$GetData =$
IF $CurTransaction \neq None$ THEN
$StatusWord := ISO\_Error \ || \ CurTransaction := None$
ELSE
$StatusWord := ISO\_Ok$
END;

$InitializeTransaction =$
IF $CurTransaction \neq None \ \vee \ ChannelIsSecured = \mathsf{FALSE}$ THEN
$StatusWord := ISO\_Error \ || \ CurTransaction := None$
ELSE
$StatusWord :\in StatusType$;
IF $StatusWord = ISO\_Ok$ THEN
$CurTransaction :\in \{Debit, Credit\}$
END
END;

$CompleteTransaction =$
IF $CurTransaction = None$ THEN
$StatusWord := ISO\_Error$
ELSE
$CurTransaction := None \ || \ StatusWord := ISO\_Ok$
END

END

GeneSyst: a Tool to Reason about Behavioral Aspects of B Event Specifications. Application to Security Properties*

Didier Bert, Marie-Laure Potet, and Nicolas Stouls

Laboratoire Logiciels Systèmes Réseaux - LSR-IMAG - Grenoble, France
{Didier.Bert, Marie-Laure.Potet, Nicolas.Stouls}@imag.fr

Abstract. In this paper, we present a method and a tool to build symbolic labelled transition systems from B specifications. The tool, called GeneSyst, can take into account refinement levels and can visualize the decomposition of abstract states in concrete hierarchical states. The resulting symbolic transition system represents all the behaviors of the initial B event system. So, it can be used to reason about them. We illustrate the use of GeneSyst to check security properties on a model of electronic purse.

1 Introduction

Formal methods, such as the B method [**?**], ensure that the development of an application is reliable and that properties expressed in the model are satisfied by the final program. However, they do not guarantee that this program fulfills the informal requirements, nor the needs of the customer. So, it is useful to propose several views about the specifications, in order to be sure that the initial model is suitable for the customer and that the development can continue on this basis. One of these important insights is the representation of the behavior of programs by means of diagrams (statecharts). Moreover, some particular views, if they are themselves formal, can provide new means to prove properties that cannot easily be checked in the first model.

In this paper, we present a method and a tool to extract a labelled transition system from a model written in event-B. The transition system gives a graphical view and represents symbolically all the behaviors of the B model. The method is able to take into account refinement levels and to show the correspondence between abstract and concrete systems, by means of hierarchical states.

We present also an application of this tool, namely, the verification of security properties. The security properties assert the occurrence or the absence of some particular events in some situation. They are a case of *atomicity property* of transactions. This is illustrated by an example of specification of an electronic

* This work was done in the GECCOO project of program "ACI : Sécurité Informatique" supported by the French Ministry of Research and New Technologies. It is also suported by CNRS and ST-Microelectronics by the way of a doctoral grant.

purse, called Demoney[?,?], developed in the SecSafe project [?]. This case study, written in Java Card [?], is an applet that has all the facilities required by a real electronic purse. Indeed, the purse can be debited from a terminal in a shop, credited by cash or from a bank account with a terminal in a bank or managed from special terminal in bank restricted area. Transactions are encrypted if needed and different levels of security are used depending on the actions. Demoney also supports to communicate with another applet on the card, for example, to manage award points on a loyalty plan. The specification of Demoney is public in version 0.8 [?], but the source code is copyrighted by Trusted Logic S.A.[1].

In Section 2, we recall the main features of event-B systems and refinements. We introduce a notion of behavioral semantics by the way of sequences of events. In Section 3, we define symbolic labelled transition systems (SLTS) and the links between SLTS and event-B systems are stated. In Section 4, we present the GeneSyst tool and an example of generation of SLTS dealing with the error cases in the Demoney case study. Section 5 presents security properties required in the application and shows how the GeneSyst diagrams can be used to check these properties. Then, we review related works, and we conclude the paper with some research perspectives in Section 6.

2 Event-B

2.1 General presentation

Event-B was introduced by J.-R. Abrial [?,?]. It is a formal development method as well as a specification language. In event-B, components are composed of constant declarations (SETS, CONSTANTS, PROPERTIES), state specification (VARIABLES, INVARIANT), initialisation and set of *events*. The events are defined by $e \mathrel{\widehat{=}} eBody$ where e is the name of the event and *eBody* is a *guarded* generalized substitution [?]. The events do not take parameters and do not return result values. They do not get preconditions and do terminate. Their effect is only to modify the internal state. If $\mathcal{S}$ is a component, then we denote by $Interface(\mathcal{S})$ the set of its events.

A well-typed and well-defined component is consistent if initialization *Init* establishes the invariant of the component and if each event preserves the invariant. So, using the notation $[S]R$ as the weakest precondition of R for substitution S, the consistency of a component is expressed by the proof obligations: $[Init]I$ and $I \Rightarrow [eBody]I$ for each event.

In the paper, we use the notions of before-after predicate of substitution T for variables x ($\mathsf{prd}_x(T)$) and the feasability predicate of a substitution as defined in the B-Book: $\mathsf{fis}(T) \Leftrightarrow \neg[T]false$ [?]. Finally, the notation $\langle T\rangle R$ means $\neg[T]\neg R$, that is to say, there exists a computation of T which terminates in a state verifying R.

[1] http://www.trusted-logic.fr/

2.2 Events and traces

The events have the form "$e \;\widehat{=}\; G \Longrightarrow T$" where G is a predicate, T is a generalized substitution such that $I \wedge G \Rightarrow \mathsf{fis}(T)$. Predicate G is called the *guard* of e and T is its *action*. They are respectively denoted by $Guard(e)$ and $Action(e)$. If the syntactic definition of an event $e \;\widehat{=}\; S$ does not fulfill this form, it can be built by computing $e \;\widehat{=}\; \mathsf{fis}(S) \Longrightarrow S$. Following the so-called *event-based* approach [**?**], the semantics of event-B systems can be chosen to be the set of all the valid sequences of event executions.

Definition 1 (Traces of Event-B systems) *A finite sequence of event occurrences $e_0.e_1.e_2 \ldots e_n$ is a trace of system $\mathcal{S}$ if and only if e_0 is the initialisation of $\mathcal{S}$, $\{e_1, e_2, \ldots, e_n\} \subseteq Interface(\mathcal{S})$ and $\mathsf{fis}(e_0 \;;\; e_1 \;;\; e_2 \;;\; \ldots \;;\; e_n) \Leftrightarrow true$.*

The set of all the finite traces of a system $\mathcal{S}$ is called $Traces(\mathcal{S})$. For the initialisation, one can notice that $\mathsf{prd}_x(Init)$ does not depend on the initial values of the variables and that $Guard(Init) \Leftrightarrow true$. The following property characterizes traces by the existence of intermediary states x_i in which the guard of e_i holds and where the pair (x_i, x_{i+1}) is in the before-after predicate of event e_i:

Property 1 (Trace characterization) *Let x be the variable space of system $\mathcal{S}$, then:* $e_0.e_1.\ldots.e_n \in Traces(\mathcal{S}) \;\Leftrightarrow$
$\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n} ([x := x_i]\, Guard(e_i) \wedge [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(Action(e_i)))$

2.3 Event-B refinement

In the event-B method, a refinement is a component called REFINEMENT. The variables can be refined (i.e. made more concrete) and a *gluing invariant* describes the relationship between the variables of the refinement and those of the abstraction. The events of refinement $\mathcal{R}$ must at least contain those of the abstraction $\mathcal{S}$ (i.e. $Interface(\mathcal{S}) \subseteq Interface(\mathcal{R})$). The other events are called *new* events.

We recall here the proof obligations of system refinements. Let I be the invariant of the abstraction $\mathcal{S}$ and J be the invariant of refinement $\mathcal{R}$, then the gluing invariant is the conjunction $I \wedge J$. The refinement is performed elementwise, that is to say, the abstract initialisation is refined by the concrete initialisation and each abstract event is refined by its concrete counterpart. Proof obligations that establish the consistency of refinements are :

For initialisation $Init$:	$[Init^R]\langle Init^S\rangle J$
For events e of $Interface(\mathcal{S})$:	$I \wedge J \Rightarrow [e^R]\langle e^S\rangle J$
For the new events ne^R :	$I \wedge J \Rightarrow [ne^R]\langle \mathsf{skip}\rangle J$

New events cannot indefinitely take the control, i.e. the refined system cannot diverge more often that the abstract one. So, a *variant* V is declared in the refined system, as an expression on a well-founded set (usually the natural numbers), and the new events must satisfy (v is a fresh variable) :

V is a natural expression : $I \wedge J \Rightarrow V \in \mathbb{N}$

New events ne^R decrease the variant : $I \wedge J \Rightarrow [v := V][ne^R](V < v)$

Finally, a proof obligation of *liveness preservation* is usually required. If $\mathcal{S}$ contains m events and $\mathcal{R}$ contains p new events, then:

$$I \wedge J \Rightarrow (\bigvee_{i=1}^{m} Guard(e_i^S) \Rightarrow (\bigvee_{i=1}^{m} Guard(e_i^R) \ \vee \ \bigvee_{i=1}^{p} Guard(ne_i^R)))$$

Traces associated to refinements are defined as for the systems.

3 Symbolic labelled transition systems associated to B systems

3.1 Symbolic transition systems

We define *symbolic* labelled transition systems:

Definition 2 (Symbolic labelled transition system) *A symbolic labelled transition system (SLTS) is a 4-uple $(N, Init, U, W)$ where*
- *N is a set of states, and $Init$ is the initial state ($Init \in N$)*
- *U is a set of labels of the form (D, A, e), where D and A are predicates and e is an event name*
- *W is a transition relation $W \subseteq \mathbb{P}(N \times U \times N)$.*

A transition $(E, (D, A, e), F)$ means that, in state E, the event e is enabled if D holds and, starting from state E, if event e is enabled, then it reaches state F if A holds. Predicate D is called the *enabledness* predicate and A is called the *reachability* predicate.

States N are interpreted as subsets of variable spaces on variables x. So, the interpretation of N is given by a function $\mathcal{I}$ such that $\mathcal{I}(E)$ is a predicate on free variables x which characterizes the subset represented by E. In the next definition, we determine the actual conditions to cross a transition from a particular state value x_1 of E_1 to x_2 of E_2 by an event e which is defined in an event-B system $\mathcal{S}$. For that, e must be enabled in x_1, x_2 must be reachable from x_1 by e, and (x_1, x_2) must belong to the before-after predicate of e:

Definition 3 (Transition crossing) *Let $(E_1, (D, A, e), E_2)$ be a transition of a SLTS $\mathcal{T}$ on a system $\mathcal{S}$, and given x_1 and x_2 some values of the state variables x which satisfy the invariant of $\mathcal{S}$, then a crossing from x_1 to x_2 by this transition is legal if and only if :*
1. $[x := x_1](\mathcal{I}(E_1) \wedge D \wedge A)$
2. $[x, x' := x_1, x_2]\, \mathsf{prd}_x(Action(e))$
3. $[x := x_2]\mathcal{I}(E_2)$

Such a legal transition crossing is denoted by :

$$(E_1, x_1) \rightsquigarrow^{(D,A,e)} \rightsquigarrow \ (E_2, x_2)$$

Now, we introduce the notion of *path* in a symbolic labelled transition system. A path is a sequence of event occurrences, starting from the initial state, which goes over a transition system through legal transition crossings.

Definition 4 (Paths) *Given a symbolic labelled transition system $\mathcal{T}$ on a system $\mathcal{S}$, a sequence of event occurrences $e_0.\ldots.e_{n+1}$ is a path in $\mathcal{T}$ if there exists a list of states $E_0, \ldots, E_{n+1}$ of N, with $E_0 = Init_{\mathcal{T}}$, and a list of transitions $(D_i, A_i, e_i), i \in 0..n$, such that :*

$$\exists x_0, \ldots, x_{n+1} \cdot (\bigwedge_{i=0}^{n}((E_i, x_i) \rightsquigarrow^{(D_i, A_i, e_i)} (E_{i+1}, x_{i+1})))$$

The set of all the finite paths of $\mathcal{T}$ is called $Paths(\mathcal{T})$.

3.2 Construction of states and transitions

The aim of this section is to show how to compute a SLTS, from an event-B system $\mathcal{S}$ and given a set of states N. First, to build the states N, consider a list of predicates $\{P_1, \ldots, P_n\}$ on the variable space. We require that this set is *complete* with respect to the invariant, i.e. all the states specified by the invariant are included in the states determined by the P_i predicates, i.e.

$$I \;\Rightarrow\; \bigvee_{i=1}^{n} P_i$$

Then, the states of the SLTS are $N = \{Init_{\mathcal{S}}, E_1, \ldots, E_n\}$ with the interpretation defined by:

$$\mathcal{I}(Init_{\mathcal{S}}) = true \qquad\qquad \mathcal{I}(E_i) = P_i \;\wedge\; I, \quad i \in 1..n$$

We denote by $N1$ the set $N - \{Init_{\mathcal{S}}\}$. From the completeness property above and the definition of N, we get: $I \;\Leftrightarrow\; \bigvee_{i=1}^{n} \mathcal{I}(E_i)$.

Now, we express the conditions to ensure that a symbolic labelled transition system $\mathcal{T}$ represents the same set of behaviors as the associated system $\mathcal{S}$. For that, in a starting state E, the enabledness condition must be equivalent to the guard of the event e, and if the target state is F, the reachability condition must be equivalent to the possibility to reach F through e, when the enabledness predicate holds, so the condition:

Condition 1 (Valid transitions) *Let $\mathcal{S}$ be a system, E and F two states in N as defined above, and e an event, then the transition (E,(D,A,e),F) is valid if and only if predicates D and A satisfy :*

a) $\mathcal{I}(E) \;\Rightarrow\; (D \Leftrightarrow Guard(e))$

b) $\mathcal{I}(E) \wedge Guard(e) \;\Rightarrow\; (A \Leftrightarrow \langle Action(e)\rangle \mathcal{I}(F))$

Notice that, by applying the definition of the conjugate weakest precondition, condition *b)* is equivalent to :

$$\mathcal{I}(E) \wedge Guard(e) \;\Rightarrow\; (A \Leftrightarrow \exists x' \cdot (\mathsf{prd}_x(Action(e)) \wedge [x := x']\mathcal{I}(F)))$$

A SLTS with all the transitions valid with respect to a system $\mathcal{S}$ is called a valid symbolic labelled transition system.

Theorem 1 (Traces and paths equality) *Let $\mathcal{S}$ be an event-*B *system with invariant I and events Ev and let $\mathcal{T}$ be a valid symbolic labelled transition system built from $\mathcal{S}$, then:*

$$Traces(\mathcal{S}) = Paths(\mathcal{T})$$

Proof: We prove that, for all t, $t \in Paths(\mathcal{T}) \Leftrightarrow t \in Traces(\mathcal{S})$.
The path $t \ \widehat{=}\ e_0.e_1.\ldots.e_n$ is a path for the state sequence $E_0, E_1, \ldots, E_{n+1}$ iff (Definition 4): $\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n}((E_i, x_i) \rightsquigarrow^{(D_i, A_i, e_i)} (E_{i+1}, x_{i+1}))$
By using Definition 3, we get:

$$\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n}([x := x_i](\mathcal{I}(E_i) \wedge D_i \wedge A_i)$$
$$\wedge\ [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(Action(e_i))\ \wedge\ [x := x_{i+1}]\mathcal{I}(E_{i+1}))$$

By Condition 1, one can replace D_i by $Guard(e_i)$ and A_i by $\exists x' \cdot (\mathsf{prd}_x(Action(e_i)) \wedge [x := x']\mathcal{I}(E_{i+1}))$. The formula above is simplified and becomes:

$$\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n}([x := x_i](\mathcal{I}(E_i) \wedge Guard(e_i)) \quad (1)$$
$$\wedge\ [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(Action(e_i))\ \wedge\ [x := x_{i+1}]\mathcal{I}(E_{i+1}))$$

We must prove that this formula is equivalent to the characterization of the traces (Property 1):

$$\exists x_0, \ldots, x_{n+1} \cdot \bigwedge_{i=0}^{n}([x := x_i]Guard(e_i) \quad (2)$$
$$\wedge\ [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(Action(e_i))\ \wedge\ [x := x_{i+1}]I)$$

Implication (1) $\Rightarrow$ (2) is verified because states E_i are such that $\mathcal{I}(E_i) \Rightarrow I$ (Section 3.2). To prove (2) $\Rightarrow$ (1), we must exhibit a list of states $E_0, E_1, \ldots, E_{n+1}$ such that these states satisfy (1). This follows from the fact that $\mathcal{I}(E_0) = true$ and from $I \Rightarrow \bigvee_{i=1}^{n} \mathcal{I}(E_i)$, which ensures that one of the states $\mathcal{I}(E_i)$ necessarily holds when I hold. □

3.3 Labelled transition systems for the refinements

We propose now the construction of a symbolic labelled transition system for the refinements. Our aim is to highlight the links between abstract and concrete transition systems, while preserving the overall structure of the abstract system. One aspect of the refinement is the change of the variable representation and redefinition of the events of the abstraction, according to the new representation. The point is taken into account by the notion of *state projection*.

In the following, $\mathcal{S}$ is a specification, $\mathcal{R}$ is its refinement with gluing invariant L, and $\mathcal{T}^S$ is a symbolic labelled transition system for $\mathcal{S}$. States E^S and F^S are states in $\mathcal{T}^S$. We assume that the variable set x^S of $\mathcal{S}$ is disjoint to the variable set x^R of the refinement. If some variables of the specification are kept in the refinement, they can be renamed and an equality between both variables is added to the invariant.

Definition 5 (State projection) *Let $\mathcal{S}$ be a system with variables x^S and $\mathcal{R}$ be the refinement of $\mathcal{S}$ according to L. A state E^R of $\mathcal{T}^{\mathcal{R}}$, $E^R \neq Init_{\mathcal{R}}$ is the projection of E^S of $\mathcal{T}^{\mathcal{S}}$, denoted by $E^R = Proj_L(E^S)$, iff:*

$$\mathcal{I}(E^R) \Leftrightarrow \exists x^S \cdot (L\ \wedge\ \mathcal{I}(E^S))$$

We propose to build a SLTS, called $Proj_L(\mathcal{T}^S)$, in which states are automatically deduced from abstract states and gluing invariant. The SLTS projection $Proj_L(\mathcal{T}^S)$ of the refinement $\mathcal{R}$ of system $\mathcal{S}$ with gluing invariant L is such that: the initial state is any q_0 with $\mathcal{I}(q_0) = true$; the other states of the projection are the projections of abstract states, i.e. $N1^R = \{Proj_L(q) \mid q \in N1^S\}$. The transitions are $(E^R, (D', A', e^R), F^R)$ where $e^R \in \mathcal{R}$ and D', A' are such that Condition 1 is satisfied. A transition $(E^R, (D', A', e^R), F^R)$ is said a *projection of transition* $(E^S, (D, A, e^S), F^S)$ iff $E^R = Proj_L(E^S)$, $F^R = Proj_L(F^S)$ and event e^R is the refinement of e^S. By construction, $Paths(Proj_L(\mathcal{T}^S)) = Traces(\mathcal{R})$. This equality can be proved in the same way as in Theorem 1.

Property 2 (Transition projection) *With the definitions above, let* $(E^R, (D', A', e^R), F^R)$ *be the projection of transition* $(E^S, (D, A, e^S), F^S)$*, then we have:*

$$\mathcal{I}(E^S) \ \wedge \ L \ \wedge \ D' \Rightarrow D$$

This property says that any transition enabled from a state $Proj_L(E^S)$ in a refinement $\mathcal{R}$ actually must be enabled in specification $\mathcal{S}$ (if the refinement is proved correct). Property 2 can make the computation of the transitions simpler. Indeed, if $e \in Interface(\mathcal{S})$, then, for all the transitions (E^S, e, F^S) of the abstraction, it is only necessary to examine the transitions $(Proj_L(E^S), e, E')$ with $E' \in N1^R$. No other transition can be labelled by e from this state.

Another key aspect of refinement is the refinement of behaviors. New events may be introduced that make the actions more detailed. These new events are not observable at the abstract level, as the stuttering in TLA [**?**]. Very often, new variables are introduced. Thus, it is useful to visualize the states referring to these variable changes. In order to preserve the structure of the abstract system, we choose to refine each abstract state in an independent way. So, the transitions, relative to events which belong to $Interface(\mathcal{S})$, are preserved by the introduction of hierarchical states.

Definition 6 (Hierarchical states) *A set of sub-states* $\{E_1^R, \ldots, E_m^R\}$ *can be associated to a super-state* $Proj_L(E^S)$ *of* $\mathcal{R}$ *if and only if*

$$\bigvee_{i=1}^{m} \mathcal{I}(E_i^R) \Leftrightarrow \mathcal{I}(Proj_L(E^S))$$

In a refined system, the user must decide what projections of abstract states are decomposed and s/he must provide the predicates of the decomposition. If the abstract states are disjoint, then the transitions associated to the new events appear only between the sub-states of a hierarchical state. An example of refinement with decomposition of states is given in Section 4.4.

4 The GeneSyst tool

4.1 Presentation

The GeneSyst tool is intended to generate a symbolic labelled transition system $\mathcal{T}$ from an event-B system $\mathcal{S}$ and a set of states N. Such a generated SLTS

will be denoted by $\mathcal{T}(\mathcal{S}, N)$. The input of the tool is a B component, where the ASSERTIONS clause contains the formula $P_1 \vee \ldots \vee P_n$, which characterizes the list of predicates $\{P_1, \ldots, P_n\}$. By this way, the condition of completeness (section 3.2) is generated as proof obligation.

We give a sketch of the algorithm which computes the transitions of $\mathcal{T}(\mathcal{S}, N)$: it uses three main variables: the set of visited states, *visited*, the set of processed states, *processed*, and the set of computed transitions tr. First, the initial state is put in the *visited* set. Then each state E in the *visited* set is processed: this consists in computing the transitions $(E, (D, A, e), F)$ with all events e to all non-initial states F of the system. Predicates D and A are determined following the algorithm defined in the following section. If D or A are not *false* then the transition $(E, (D, A, e), F)$ is added to tr, and if F has not been processed, it is put in the *visited* set. After the processing of state E, E is removed from *visited* and put in set *processed*. When *visited* is empty, then tr contains all the computed transitions of $\mathcal{T}(\mathcal{S}, N)$ and *processed* contains the set of reachable states. The algorithm terminates, because the set of states to be visited is finite (bounded by the cardinal of N). This algorithm guarantees that the resulting SLTS is a valid transition system for $\mathcal{S}$, with given states N.

4.2 Proof obligations

A subprocedure of the algorithm is to determine effectively the enabledness predicate and the reachability predicate, given a triple (E, e, F). For sake of usability of the resulting transition system, it is interesting to examine three cases: predicates are *true*, *false* or other. This information can be obtained by proof obligations. In Fig. 1, we give the conditions for the calculus of these predicates. Obviously, if D and/or A is *false*, then the transition is not possible.

	Proof obligations	D for (E, e, F)
(1)	$\forall x \cdot (\mathcal{I}(E) \Rightarrow \mathit{Guard}(e))$	true
(2)	$\forall x \cdot (\mathcal{I}(E) \Rightarrow \neg \mathit{Guard}(e))$	false
(3)	$\exists x \cdot (\mathcal{I}(E) \wedge \mathit{Guard}(e))$	$\mathit{Guard}(e)$
	Proof obligations	A for (E, e, F)
(4)	$\forall x \cdot (\mathcal{I}(E) \wedge \mathit{Guard}(e) \Rightarrow \langle \mathit{Action}(e) \rangle \mathcal{I}(F))$	true
(5)	$\forall x \cdot (\mathcal{I}(E) \wedge \mathit{Guard}(e) \Rightarrow [\mathit{Action}(e)] \neg \mathcal{I}(F))$	false
(6)	$\exists x \cdot (\mathcal{I}(E) \wedge \mathit{Guard}(e) \wedge \langle \mathit{Action}(e) \rangle \mathcal{I}(F))$	$\langle \mathit{Action}(e) \rangle \mathcal{I}(F)$

Fig. 1. Proof obligations for enabledness and reachability

In practice, the GeneSyst tool computes the proof obligations (POs) above and interacts with AtelierB to discharge the POs. For each triple (E, e, F):

1. if proof obligation (1) is automatically discharged then D is *true*.
2. if proof obligation (2) is automatically discharged then D is *false* and transition (E, e, F) does not occur in the resulting $\mathcal{T}(\mathcal{S}, N)$.
3. otherwise, D is $\mathit{Guard}(e)$ by default.

Then, after cases 1. and 3., GeneSyst computes the proof obligations for determining the reachability predicate A.

4. if proof obligation (4) is automatically discharged then A is *true*.
5. if proof obligation (5) is automatically discharged then A is *false* and transition (E, e, F) does not occur in the resulting $\mathcal{T}(\mathcal{S}, N)$.
6. otherwise, the transition is kept with $\langle Action(e)\rangle\, \mathcal{I}(F)$ as A, by default.

We can notice that Condition 1 about the validity of the transitions is well satisfied by construction. The *by default cases* in 3. and 6. correspond to several possibilities. Either there exist values in state E for which the transition is crossable (guard of e is true and state F is reachable), or there are not (the guard is false or state F is not reachable). However, in both possibilities, these transitions are included in the resulting transition system. To manage this feature, we define the notion of *minimal* symbolic labelled transition system.

Definition 7 (Minimal SLTS) *A minimal SLTS is a SLTS where all the transitions are valid, i.e. satisfy a) and b) of Condition 1, and also satisfy:*

c) $D \not\Leftrightarrow false$ *and* $A \not\Leftrightarrow false$

A SLTS built by GeneSyst is minimal if all the proof obligations of D and A have been effectively discharged in step 1. or 2. and step 4. or 5. in the algorithm above. To minimize the number of by-default transitionss, we have designed two variants of the algorithm. The first optional alternative of the algorithm is to change cases 3. and 6. into:

3'. if proof obligation (3) is automatically discharged, then D is $Guard(e)$ by proof, otherwise, D is $Guard(e)$ by default.
6'. if proof obligation (6) is automatically discharged, then A is $\langle Action(e)\rangle\, \mathcal{I}(F)$ by proof, otherwise, the transition is kept with $\langle Action(e)\rangle\, \mathcal{I}(F)$ as A by default.

Another option of the tool allows the user to get the POs which have not been automatically discharged. Then, s/he can do an interactive proof to complete the work and return the information that the PO is discharged or not. However, the interactive mode is not very practicable when there are a great number of proof obligations that are not automatically discharged. It becomes useful to check actually the absence of some critical transitions (cases 2. and 4.).

4.3 Transition systems associated to the Demoney case study

In Fig. 2, we give an example of transition system generated from a subset of the abstract specification of the Demoney case study. The B machine is provided in appendix. We just have represented four methods imposed by the Demoney specification [**?**]: *InitializeTransaction*, *CompleteTransaction*, *Reset* and *GetData*. The two methods *InitializeTransaction* and *CompleteTransaction* have

to be executed in sequence. If they are called in the wrong order then an error must be returned. Moreover, any other methods cannot be invoked between them, except the method *Reset* which models the extraction of the card from the terminal. If it is called during a transaction, all the internal variables must be restored at their initial values. Finally, method *GetData* has been defined to represent any other method which plays a neutral rôle with respect to transactions.

Let us notice that our model has been expressed with events. In the applet Demoney, methods have neither parameters nor result, because they communicate through a global variable, named $APDU$, which allows the information transfert between the card and the terminal. An error can be returned by means of the same variable. Finally, methods have no precondition, because they are callable at any time. So the transformation of methods in events is straightforward.

In the diagrams generated by GeneSyst, transitions are prefixed by the information about predicates D and A. A predicate denoted by "[ ]" means *true*, while "[G]" means that the transition is computed by cases 3. or 6. (see section 4.2).

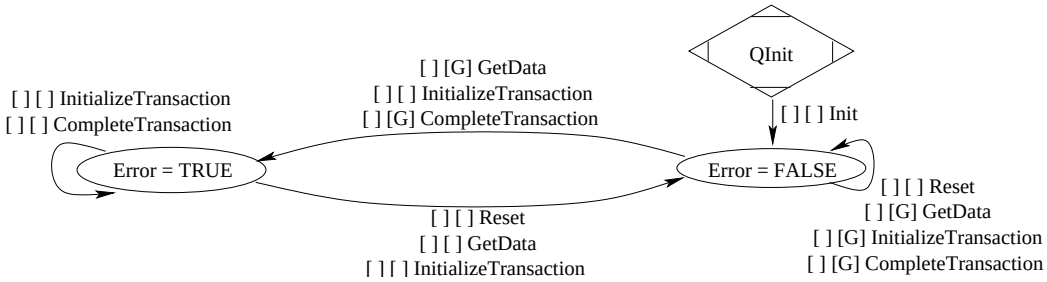

Fig. 2. Transition system associated to the error detection in the Demoney specification

Fig. 2 points out cases in which errors can occur. Transitions have no enabledness condition, because all the guards are *true* in the model. Some reachability conditions do not reduce to *true*, as for the event *GetData*, which is defined by:

$$\begin{aligned} GetData = \ & \text{IF } EngagedTrans = \mathsf{TRUE} \text{ THEN} \\ & \quad Error := \mathsf{TRUE} \parallel EngagedTrans := \mathsf{FALSE} \\ & \text{ELSE } Error := \mathsf{FALSE} \text{ END;} \end{aligned}$$

From state $Error = \mathsf{FALSE}$, event *GetData* can reach the state $Error = \mathsf{TRUE}$ with the condition $EngagedTrans = \mathsf{TRUE}$ and stays in $Error = \mathsf{FALSE}$ otherwise. Let us remark also that *GetData* is enabled in state $Error = \mathsf{TRUE}$ and always reaches state $Error = \mathsf{FALSE}$ because of the invariant $Error = \mathsf{TRUE} \Rightarrow EngagedTrans = \mathsf{FALSE}$.

4.4 Transition system associated to a refinement of Demoney

In our refinement of Demoney, the boolean variable *Error* is changed into a value of a given set *StatusType*, which intends to describe error codes, as imposed by

the specification [**?**]. In the same way, the boolean variable *EngagedTrans* is refined into a value of a given set *TransactionType*, which indicates the exact type of the current transaction. Finally, we have introduced the channel with two levels of security (FALSE and TRUE). All this information is declared in the invariant below (see also the refinement in appendix):

INVARIANT

$$
\begin{array}{l}
\quad StatusWord \in StatusType \ \wedge \ CurTransaction \in TransactionType \ \wedge \\
\quad ChannelIsSecured \in \mathsf{BOOL} \ \wedge \\
\quad ((Error = \mathsf{FALSE}) \Leftrightarrow (StatusWord = ISO\_Ok)) \ \wedge \\
\quad ((EngagedTrans = \mathsf{FALSE}) \Leftrightarrow (CurTransaction = None)) \ \wedge \\
\quad ((CurTransaction \neq None) \Rightarrow (ChannelIsSecured = \mathsf{TRUE})) \ \wedge \\
\quad ((StatusWord \neq ISO\_Ok) \Rightarrow (CurTransaction = None))
\end{array}
$$

Fig. 3 is built from this refinement. State $Error = \mathsf{FALSE}$, which corresponds to $StatusWord = ISO\_Ok$, is split into two states according to that a transaction is engaged or not.

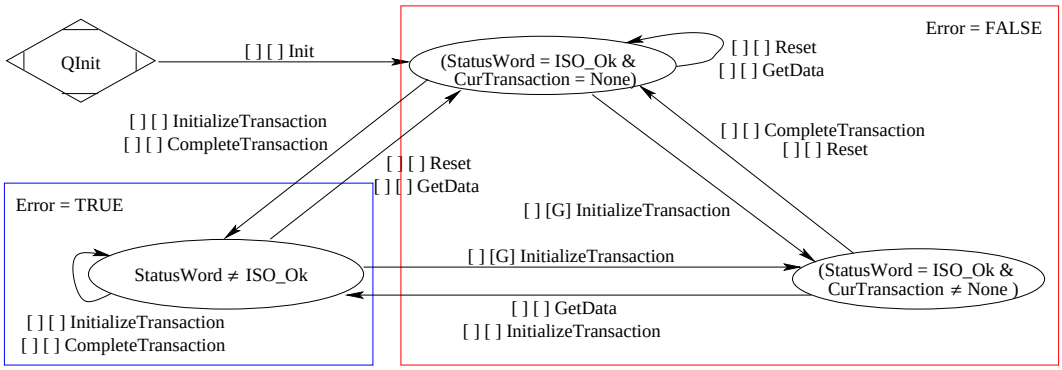

Fig. 3. Transition system associated to the refinement of the error detection

As expressed in Definition 6, the predicate given to GeneSyst to describe the states has to be a conjonction of equivalences between an abstract state and a disjonction of refined states. This predicate is written in the assertion clause. For example, the assertion below has been used to generate Fig. 3.

$$
\begin{array}{l}
((Error = \mathsf{TRUE}) \Leftrightarrow ((StatusWord \neq ISO\_Ok \ \wedge \ CurTransaction = None) \\
\qquad\qquad\qquad\qquad \vee \ (StatusWord \neq ISO\_Ok \ \wedge \ CurTransaction \neq None))) \\
\ \wedge \\
((Error = \mathsf{FALSE}) \Leftrightarrow ((StatusWord = ISO\_Ok \ \wedge \ CurTransaction = None) \\
\qquad\qquad\qquad\qquad \vee \ (StatusWord = ISO\_Ok \ \wedge \ CurTransaction \neq None)))
\end{array}
$$

With the splitting of the state $Error = \mathsf{FALSE}$, transition conditions are simplified in *true* or *false* or, in the worst case, are unchanged. For example, in Fig. 2, the transition labelled by $[][G]CompleteTransaction$ and going from

$Error = \mathsf{FALSE}$ to $Error = \mathsf{TRUE}$ is, in Fig. 3, going from $StatusWord = ISO\_Ok \wedge CurTransaction = None$ to $StatusWord \neq ISO\_Ok \wedge CurTransaction = None$ with the label $[][]CompleteTransaction$. So, its reachability has been made more precise. The same effect occurs on transition $[][G]CompleteTransaction$ going from $Error = \mathsf{FALSE}$ to $Error = \mathsf{FALSE}$, which is refined by $[][]CompleteTransaction$ going from $CurTransaction \neq None$ to $CurTransaction = None$ in the super-state $Error = \mathsf{FALSE}$. These two specializations are directly due to the introduction of the $CurTransaction$ variable.

5 Verification of Security Properties on Demoney

In this section we propose a formalism to express properties relative to security aspects and we show how GeneSyst can be used to verify these properties. We will next give a concrete example relative to the Demoney case study.

5.1 Properties

Generally, security is designed and implemented through different levels of abstraction. Security policies are defined by a set of rules according to which the system can be regulated, in order to guarantee expected properties, as confidentiality or integrity. Security policies are then implemented through software and hardware functions, called security mechanisms. Such an approach has been adopted by the **C**ommon **C**riteria norm [**?**] which proposes, through the notion of assurance requirements, a catalogue of security policies and a hierarchy of mechanisms.

In this paper we focus on security properties relative to constraints on the global behavior of the system, as authentication procedures or access control. In this case, security requirements can be seen as constraints on the execution order of atomic actions, as operation calls. F. Schneider claims in [**?**] that automata are a well-adapted formalism which can, both, be used to specify some forms of security policies and to control implementations during their execution. On the other hand, K. Trentelman and M. Huisman [**?**] propose a logic that can be used also to express some forms of security properties, as temporal properties on **JML** specifications.

We adopt a formalism based on logic formulas, which allows us to point out expected behaviors either in specifying correct executions, or in specifying security violations. That offers a good flexibility and is suitable to describe as well open policies as closed policies, respectively relative to negative authorizations and positive authorizations [**?**].

5.2 Predicates of security properties

Security properties are often represented as a list of first order logic formulas that have to be verified. We want to define some predicates to make the expression of these formulas easier. Predicates that we introduce express the ability of an

event to start from a state (*Enabled* and *AlwaysEnabled*) and the existence of a transition between two states (*Crossable* and *AlwaysCrossable*).

Definition 8 (*Enabled*, *AlwaysEnabled*, *Crossable* and *AlwaysCrossable*) *Given p_1 and p_2 two state predicates and an event ev from a system $\mathcal{S}$ with variables x, then:*

$$\begin{array}{lcl} Enabled(p_1, ev) & \hat{=} & \exists x \cdot (p_1 \ \wedge \ Guard(ev)) \\ AlwaysEnabled(p_1, ev) & \hat{=} & \forall x \cdot (p_1 \Rightarrow Guard(ev)) \\ Crossable(p_1, ev, p_2) & \hat{=} & \exists x \cdot (p_1 \ \wedge \ \langle ev \rangle p_2) \\ AlwaysCrossable(p_1, ev, p_2) & \hat{=} & \forall x \cdot (p_1 \Rightarrow [ev] p_2) \end{array}$$

Let us note that if $Enabled(p_1, ev) \Leftrightarrow false$, then, for each predicate p_2, $AlwaysCrossable(p_1, ev, p_2)$ will be *true* instead of *false*, which is the intuitive value expected. In the same way, if p_1 is equivalent to *false* then *AlwaysEnabled* and *AlwaysCrossable* are always *true*. Moreover, we can notice that:

$$Crossable(p_1, ev, p_2) \Rightarrow Enabled(p_1, ev)$$

From this definition we can deduce the properties below, relative to the implication:

Property 3 *Given p_1, p_2 and p_3 three predicates and an event ev then:*

- *if* $p_1 \Rightarrow p_3$ *and* $Enabled(p_1, ev)$ *then* $Enabled(p_3, ev)$
- *if* $p_3 \Rightarrow p_1$ *and* $AlwaysEnabled(p_1, ev)$ *then* $AlwaysEnabled(p_3, ev)$
- *if* $p_1 \Rightarrow p_3$ *and* $Crossable(p_1, ev, p_2)$ *then* $Crossable(p_3, ev, p_2)$
- *if* $p_2 \Rightarrow p_3$ *and* $Crossable(p_1, ev, p_2)$ *then* $Crossable(p_1, ev, p_3)$
- *if* $p_3 \Rightarrow p_1$ *and* $AlwaysCrossable(p_1, ev, p_2)$ *then* $AlwaysCrossable(p_3, ev, p_2)$
- *if* $p_2 \Rightarrow p_3$ *and* $AlwaysCrossable(p_1, ev, p_2)$ *then* $AlwaysCrossable(p_1, ev, p_3)$

Here are two examples:

Reactivity of a system. The **JavaCard** specification imposes that any APDU instruction is callable at any time. Given $\mathcal{S}$ a system and I its invariant, then this formula can be expressed as follows:

$$\forall ev \cdot (ev \in Interface(\mathcal{S}) \Rightarrow AlwaysEnabled(I, ev))$$

Unicity of the ways to reach a state. In some cases, like access control, we want to impose that the only way to reach a state P is to execute a particular event *Begin*. If I is the invariant of $\mathcal{S}$, then this property can be expressed as follows:

$$\forall ev \cdot (ev \in Interface(\mathcal{S}) \ \wedge \ ev \neq Begin \Rightarrow AlwaysCrossable(I, ev, \neg P))$$

5.3 Property checking using GeneSyst SLTS

Security properties could be verified on B specifications, using definition 8. Nevertheless, in some cases, the SLTS produced by GeneSyst can be directly exploited. Then, the verification consists in using syntactic information relative to enabledness and reachability of transitions. Properties 4–7 list the different cases where the predicates above can be directly established from a symbolic labelled transition system.

Properties 4–7 share the following hypothesis: *Given an event e and q_1, q_2 two states from a SLTS $\mathcal{T}$, such as $\mathcal{I}(q_1) \not\Rightarrow false$ and $(q_1, (D, A, e), q_2) \in W_{\mathcal{T}}$, then predicates Enabled, AlwaysEnabled, Crossable and AlwaysCrossable can be determined as follows:.*

Property 4 (Enabledness condition - general case)

1. $D \equiv true$	$\Rightarrow$	$Enabled(q_1, e)$
2. $D \equiv false$	$\Rightarrow$	$\neg Enabled(q_1, e)$
3. $D \equiv true$	$\Rightarrow$	$AlwaysEnabled(q_1, e)$
4. $D \equiv false$	$\Rightarrow$	$\neg AlwaysEnabled(q_1, e)$

If the SLTS used to verify the property is minimal, then Property 4 can be enlarged: the conditions are necessary (and sufficient) and conditions 1 and 4 are refined.

Property 5 (Enabledness for minimal SLTS)

1. $D \not\equiv false$	$\Leftrightarrow$	$Enabled(q_1, e)$
2. $D \equiv false$	$\Leftrightarrow$	$\neg Enabled(q_1, e)$
3. $D \equiv true$	$\Leftrightarrow$	$AlwaysEnabled(q_1, e)$
4. $D \not\equiv true$	$\Leftrightarrow$	$\neg AlwaysEnabled(q_1, e)$

In the same way, syntactic conditions to check *Crossable* and *AlwaysCrossable* predicates are:

Property 6 (Reachability condition - general case)

5. $A \equiv true$	$\Rightarrow$	$Crossable(q_1, e, q_2)$
6. $A \equiv false \ \vee \ D \equiv false$	$\Rightarrow$	$\neg Crossable(q_1, e, q_2)$
7. $A \equiv true \ \wedge$ $\forall q_i \cdot (q_2 \not\equiv q_i \Rightarrow (q_1, (D, A_2, e), q_i) \notin W_{\mathcal{T}})$	$\Rightarrow$	$AlwaysCrossable(q_1, e, q_2)$
8. $A \equiv false$	$\Rightarrow$	$\neg AlwaysCrossable(q_1, e, q_2)$

Cases 7 and 8 are not symetric, as it would be expected, because, syntacticaly, we can just compare names of states, not the intersection of their interpretation. Just as for enabledness, the conditions can be enlarged, when the SLTS is minimal, as follow:

Property 7 (Reachability for minimal SLTS)

5. $A \not\equiv false$	$\Leftrightarrow$	$Crossable(q_1, e, q_2)$
6. $A \equiv false \ \vee \ D \equiv false$	$\Leftrightarrow$	$\neg Crossable(q_1, e, q_2)$
8. $A \not\equiv true$	$\Rightarrow$	$\neg AlwaysCrossable(q_1, e, q_2)$

Cases 7 and 8 are just sufficient conditions because of the limitation of the syntactic verification. Case 7 is not present in Property 7 because it is the same as in Property 6 Finally, Property 3 allows the deduction of derived properties from the four properties above, by weakening or strenghtening the states.

5.4 Example of a property checking

In this section, we develop a real example of Demoney property and we do its verification by using the SLTS given in Figure 3. In the Demoney specification [**?**], the two APDU instructions *InitializeTransaction* and *CompleteTransaction* have to be executed in sequence, without any other instructions between them and without reaching any error state, to make a transaction. However, the card can be withdrawn at any time (modelled by the *Reset* event) without generating any error. Transaction atomicity property can be decomposed in five formulas given below, where I stands for the invariant of the Demoney specification. Moreover, SLTS of Figure 3 is minimal and events are always enabled from all state of the SLTS. Finally, note than the invariant I is equivalent to the union of all state predicates (Section 3.2).

Formula 1: There exists at least a value in I such that the event *Initialize-Transaction* can reach $CurTransaction \neq None$:

$$Crossable(I, InitializeTransaction, CurTransaction \neq None)$$

Predicate $CurTransaction \neq None$ directly corresponds to a state predicate. Since there exists a transition from $CurTransaction = None \ \wedge \ StatusWord = ISO\_Ok$ to $CurTransaction \neq None$, labelled with $[\,][G]InitializeTransaction$, then we can use case 5 of Property 7 and conclude that the Formula 1 is *true*.

Formula 2: For all values, the event *InitializeTransaction* goes into the state $CurTransaction \neq None$ or into an error state:

$$AlwaysCrossable(I, InitializeTransaction, \\ CurTransaction \neq None \ \vee \ StatusWord \neq ISO\_Ok)$$

$CurTransaction \neq None$ and $StatusWord \neq ISO\_Ok$ are two state predicates, and all the transitions labelled with *InitializeTransaction* go only in one of these states. Then, due to case 7 of property 6, this formula is *true*.

Formula 3: From $CurTransaction \neq None$, all events, but $CompleteTransaction$ and *Reset*, go to an error state:

$$\forall e \cdot (e \in Interface(\mathcal{S}) \ \wedge \ e \neq CompleteTransaction \ \wedge \ e \neq Reset \Rightarrow \\ AlwaysCrossable(CurTransaction \neq None, e, StatusWord \neq ISO\_Ok)$$

The two predicates correspond to state predicates and the only events which go elsewhere than $StatusWord \neq ISO\_Ok$ from $CurTransaction \neq None$ are *CompleteTransaction* and *Reset*. Thus Formula 4 is *true* (case 7 of Property 6).

Formula 4: Except *InitializeTransaction*, no event can reach $CurTransaction \neq None$:

$$\forall e \cdot (e \in Interface(\mathcal{S}) \ \wedge \ e \neq InitializeTransaction \Rightarrow \\ AlwaysCrossable(I, e, CurTransaction = None))$$

Predicate $CurTransaction = None$ is the union of two existing state predicates. So, we have to check if there exists an event, different from *InitializeTransaction*, that can reach $CurTransaction \neq None$. Since it is not the case, this formula is *true* (case 7 of Property 6).

Formula 5: No transition labelled by *CompleteTransaction* or *Reset* is reflexive on state $CurTransaction \neq None$:

$\neg Crossable(CurTransaction \neq None, CompleteTransaction, CurTransaction \neq None)$

and $\neg Crossable(CurTransaction \neq None, Reset, CurTransaction \neq None)$

$CurTransaction \neq None$ corresponds to a state predicate and no *CompleteTransaction* or *Reset* reflexive transition occurs. Thus this formula is *true* (case 5 of Property 7).

The model of Demoney is thus correct relatively to the atomicity security property of transactions. However, during the realisation of this example, which is a simplified Demoney applet, we found three errors due to an erroneous simplification of our complete model of Demoney.

The originality of this approach is to have brought back, under some hypotheses, the verification of security properties to a syntactic checking. However, it is important to be careful about the real value of the crossing conditions generated by GeneSyst. Indeed, if some proof obligations are not (automatically) discharged, the transitions system will have by-default transitions. Then, to properly exploit the information, we have to be sure that the property to be verify can be checked on a non-minimal SLTS.

6 Related works and Conclusion

The work presented here is in line with the ideas presented in [**?**], itself inspired by [**?**]. In [**?**], the authors propose the construction of a labelled transition system which is a finite state abstraction of the behavior of an event-B system. The existence of transitions is determined by proof obligations, as here, but the resulting transition system does not contain any information about transition crossing. Moreover, the paper does not consider the refinement step in the diagram representation.

Other work is devoted to the translation of dynamic aspects described by statecharts in the B formalism (for instance [**?**,**?**]). These approaches are inverse of ours, because they go from a diagrammatic representation to an encoding in a formal text. Their objective is to build a B model from UML descriptions. On our side, we suppose that the model has been stated and we are interested in representing the precise behavior of the system with respect to (a part of) variables, in order to check properties, or to validate the model against the requirements.

A similar approach has been envisaged for TLA [**?**] and extended in [**?**,**?**] to take in account liveness properties and refinement. As in [**?**], the generated diagrams are abstractions of the system behavior.

Several tools are dedicated to the analysis of the behavior of B components by the way of the animation of machines [**?**] or by local exhaustive model checking [**?**]. Even if some of them allow the generation of symbolic traces, these tools can

be considered as "testing" tools. They provide particular execution sequences of the system, not a static representation of all the behaviors. In [**?**], the authors describe the generation of statecharts from event-B systems, but their approach suffers from several restrictions and their diagrams are not symbolic.

In this paper, we have presented the GeneSyst tool, its logical foundations and its application to the verification of security properties. In the first part, we introduced the definition of traces of event-B systems and refinements. We formalized the notion of symbolic labelled transition systems, with transitions decorated by enabledness and reachability predicates. This gives a complete and precise view of the behavior of the system, which can be exploited for various objectives.

We described the algorithm that is implemented to generate a SLTS from a B system and a set of states, characterized by predicates. The computation of effective transitions between states is performed by proving proof obligations. Due to the indecidability of the proof process, we have the choice between two kinds of (non exclusive) results: the generation is automatic, but we can get more transitions than in the real system, or the user completes interactively the non-conclusive proofs and then, the resulting automaton reflects exactly the behavior of the system.

The user can take profit of the freedom degree achieved by the choice of the states, to obtain the best analyses useful for him/her purpose. Non classical verification techniques can be designed and implemented at this stage, to assess or to validate the model, as it was shown in the last part of the paper. This opens a large field of research in the domains of security properties, confidentiality, access control, validation of models with respect to the requirements, automatic documentation of specifications, etc. Our present research work is to develop a set of techniques in the GECCOO[2] project to express and to check security properties, as it was sketched in the paper. We want to investigate the extraction of states from the specification of property automata, the use of refinement to split states and achieve a suitable level of decomposition in order to check a property. Another work is to deal with complex B models (several refinement chains together with composition clauses SEES, INCLUDES, etc.), either by composing partial labelled transition systems, or by flattening a structured model before computing the whole associated SLTS.

[2] "Génération de Code Certifié Orienté Objet". Project of Program "ACI Sécurité Informatique", 2003.

Appendix

Machine of the Demoney specification (diagram in Fig. 2, Section 4.3):

```
MACHINE Demoney
VARIABLES
    Error, EngagedTrans
INVARIANT
    Error ∈ BOOL ∧ EngagedTrans ∈ BOOL ∧
    (Error = TRUE ⇒ EngagedTrans = FALSE) ∧
    (EngagedTrans = TRUE ⇒ Error = FALSE)
ASSERTIONS
    /* The assertion provides the states for tool GeneSyst */
    /* Here, only two states are considered according to the Error values */
    Error = FALSE ∨ Error = TRUE
INITIALISATION
    Error := FALSE || EngagedTrans := FALSE
OPERATIONS
    Reset = BEGIN EngagedTrans := FALSE || Error := FALSE END;
    GetData =
        IF EngagedTrans = TRUE THEN
            Error := TRUE || EngagedTrans := FALSE
        ELSE Error := FALSE
        END;
    InitializeTransaction =
        IF EngagedTrans = TRUE THEN
            Error := TRUE || EngagedTrans := FALSE
        ELSE
            ANY SW WHERE SW ∈ BOOL THEN
                Error := SW || EngagedTrans := bool(SW = FALSE)
            END
        END;
    CompleteTransaction =
        IF EngagedTrans = FALSE THEN
            Error := TRUE
        ELSE Error := FALSE || EngagedTrans := FALSE
        END
END
```

Refinement of the Demoney specification (diagram in Fig. 3, Section 4.4):

```
REFINEMENT Demoney_R1
REFINES Demoney
SETS
    TransactionType = {Credit, Debit, None};
    StatusType = {ISO_Error, ISO_Ok}
VARIABLES
    StatusWord, CurTransaction, ChannelIsSecured
```

INVARIANT

$StatusWord \in StatusType \ \wedge \ CurTransaction \in TransactionType \ \wedge$
$ChannelIsSecured \in \mathsf{BOOL} \ \wedge$
$((StatusWord = ISO\_Ok) \Leftrightarrow (Error = \mathsf{FALSE})) \ \wedge$
$((EngagedTrans = \mathsf{TRUE}) \Leftrightarrow (CurTransaction \neq None)) \ \wedge$
$((CurTransaction \neq None) \Rightarrow ChannelIsSecured = \mathsf{TRUE}) \ \wedge$
$((StatusWord \neq ISO\_Ok) \Rightarrow (CurTransaction = None))$

ASSERTIONS

/* Each abstract state is decomposed in two concrete states */
/* One of these states is not reachable */
$((Error = \mathsf{TRUE}) \Leftrightarrow$
$((StatusWord \neq ISO\_Ok \ \wedge \ CurTransaction = None)$
$\vee \ (StatusWord \neq ISO\_Ok \ \wedge \ CurTransaction \neq None)))$
$\wedge$
$((Error = \mathsf{FALSE}) \Leftrightarrow$
$((StatusWord = ISO\_Ok \ \wedge \ CurTransaction = None)$
$\vee \ (StatusWord = ISO\_Ok \ \wedge \ CurTransaction \neq None)))$

INITIALISATION

$StatusWord := ISO\_Ok \ || \ ChannelIsSecured := \mathsf{FALSE} \ ||$
$CurTransaction := None$

OPERATIONS

$Reset =$ BEGIN
$StatusWord := ISO\_Ok \ || \ ChannelIsSecured := \mathsf{FALSE} \ ||$
$CurTransaction := None$
END;

$GetData =$
IF $CurTransaction \neq None$ THEN
$StatusWord := ISO\_Error \ || \ CurTransaction := None$
ELSE
$StatusWord := ISO\_Ok$
END;

$InitializeTransaction =$
IF $CurTransaction \neq None \ \vee \ ChannelIsSecured = \mathsf{FALSE}$ THEN
$StatusWord := ISO\_Error \ || \ CurTransaction := None$
ELSE
$StatusWord :\in StatusType$;
IF $StatusWord = ISO\_Ok$ THEN
$CurTransaction :\in \{Debit, Credit\}$
END
END;

$CompleteTransaction =$
IF $CurTransaction = None$ THEN
$StatusWord := ISO\_Error$
ELSE
$CurTransaction := None \ || \ StatusWord := ISO\_Ok$
END

END